SYNTAX-BASED SECURITY TESTING FOR TEXT-BASED COMMUNICATION PROTOCOLS

by

Ben W. Y. Kam

A thesis submitted to the School of Computing
In conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
(April, 2010)

Copyright © Ben W. Y. Kam, 2010
Abstract

We introduce a novel Syntax-based Security Testing (SST) framework that uses a protocol specification to effectively perform security testing on text-based communication protocols. A protocol specification of a particular text-based protocol under-tested (TPUT) represents its syntactic grammar and static semantic contracts on the grammar. Mutators written in TXL break the syntactic and semantic constraints of the protocol specification to generate test cases. Different protocol specification testing strategies can be joined together to yield a compositional testing approach. SST is independent of any particular text-based protocols. The power of SST stems from the way it obtains test cases from the protocol specifications. We also use the robust parsing technique with TXL to parse a TPUT. SST has successfully revealed security faults in different text-based protocol applications such as web applications and kOganizer. We also demonstrate SST can mimic the venerable PROTOS Test-Suite: co-http-reply developed by University of Oulu.
Acknowledgements

First of all, I am deeply appreciative to my supervisor Dr. T. R Dean. His patience, endurance, encouragement, effective guidance, and generous financial support have all been indispensable for the success of my research. No thesis of this size could be completed without his help.

I would like to thank my other two supervisory committee members, Dr. James R. Cordy and Dr. Scott Knight, for their suggestions and encouragement. Without their opinions during my thesis proposal, I would never have been able to start this thesis! I also thank Dr. Ying Zou for her advice and encouragement.

I am grateful to Electrical and Computer Engineering department for awarding me a Teaching Fellowship in 2007, and the opportunity to teach a computer science course (SOFT 423 Requirements Analysis). Therefore, I was able to hone my teaching skills and make my Ph.D studies more challenging. I would also like to extend my thanks to the School of Graduate Studies for the Michael A. Jenkins Graduate Fellowship awarded to me in 2007-2008. In addition, I am appreciative to Dr. Ed Antosh, my supervisor during my B.Ed. degree at University of Windsor. He presented the theories and concepts of teaching and learning, furthering my interest in education.

I would like to thank Dr. Jamie Andrews, my supervisor during my M.Sc. degree at The University of Western Ontario. He enlightened my interest of software testing. I am thankful for the teaching enthusiasm and sincere help of my favourite professors during my undergraduate degree at Trent University; Dr. Richard T. Hurley, Dr. Wenying
Feng, and Mr. Brian F. Hircock. They provided the best study environment for me to finish my bachelor degree.

I would also like to thank Mohammed AboElFotoh, Hung Tam and Ying Yin Kwok. They were the best study partners I could have asked for. I met Mohammed during my second year of study at Queen’s University. He is the best coder that I have ever met. I met Hung when I studied a computer diploma at Seneca College in 1998. After a year, we both started a long journey of studies. This journey has been more than ten years without tiring. Ying Yin was a good academic competitor. We had a really good time finishing our final year project during our undergraduate experience. We shared one chair while working on this project for two days and nights.

I express my grateful respect to my best friends, Patrick Lam, Leo Chan, Alex Chan, Connie Ho, Chak Sing Ho, Stephen Fung, Sam Kong, Ken Tse, Polly Chan, Eric Tang, Pauline Yeung, Gary Ma, Tse Chi Ming, Tsui Wai Yip, Tsui Wai Keung, and Alex Cheung for years encouragement and support.

The most wonderful thing in my life is that I met Grace Lee. I would like to express my endless gratitude to my wife, Grace Lee, she gives me great spiritual support all the time. She always cheers me up when I felt I was at the dead end of my research. I would never have succeeded without her. Also, I would like to thank my father-in-law and mother-in-law, Mr. Calvin Lee and Mrs. Shirley Lee, my brother-in-law Philip Lee for years encouragement and support.

Finally, my deepest and sincere thank you to my mom Wai Yin Tse, my brother Wing Chung Kam, and my sister Pui Ching Kam, for their unconditional support during each step of my education as well as all the time. When I was in grade school, my brother
always taught and helped me to finish my homework. Without my sister to teach me English, I would never have been able to complete high school. Thank You all of you!
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Ben W. Y. Kam

(April, 2010)
Table of Contents

Abstract ............................................................................................................................................. i
Acknowledgements .......................................................................................................................... ii
Statement of Originality ................................................................................................................... v
Table of Contents ............................................................................................................................ vi
List of Tables .................................................................................................................................. ix
List of Figures ................................................................................................................................. xi
Glossary ........................................................................................................................................ xiv
Chapter One Introduction .............................................................................................................. 1
  1.1 Motivation and Objective ..................................................................................................... 3
  1.2 Contribution of the thesis ................................................................................................. 4
  1.3 Outline of the thesis ........................................................................................................... 5
Chapter 2 Background and Related Work ....................................................................................... 7
  2.1 Background .......................................................................................................................... 7
    2.1.1 Software testing overview ............................................................................................ 7
    2.1.2 Web application testing ............................................................................................... 9
    2.1.3 Web application vulnerabilities .................................................................................. 14
    2.1.4 Mutation testing .......................................................................................................... 19
    2.1.5 TXL source transformation ....................................................................................... 20
  2.2 Related work ........................................................................................................................ 22
Chapter 3 Syntax-based Security Testing Overview ..................................................................... 26
  3.1 Protocol dependent module ............................................................................................... 29
  3.2 Protocol independent modules ........................................................................................... 30
Chapter 4 Syntax-based Security Testing Module Anatomy ......................................................... 32
  4.1 Network sniffer ................................................................................................................... 32
  4.2 Manifest file ....................................................................................................................... 32
  4.3 Protocol specification ......................................................................................................... 33
  4.4 Markup tags ....................................................................................................................... 37
  4.5 Protocol Description Grammars ....................................................................................... 47
  4.6 Tag dictionary .................................................................................................................... 48
  4.7 Test case generator .......................................................................................................... 49
  4.8 Mutation engine ................................................................................................................. 50
Chapter 9 Conclusion and Future Work ................................................................. 132
  9.1 Conclusion ......................................................................................................... 132
  9.2 Future Work ....................................................................................................... 133
  9.2.1 Modifying the oracle ......................................................................................... 134
  9.2.2 Exploring the markup tags ............................................................................. 135
  9.2.3 Implementing an intelligent GUI user interface ............................................. 135
Appendix A Command-line commands ................................................................. 136
  A.1 To generate the httpRequest.proto ................................................................. 136
  A.2 To generate the optAndTag.grm ..................................................................... 136
  A.3 To generate the InsertMarkup.txl .................................................................. 136
  A.4 To insert markup tags onto the login packet ................................................ 136
References ............................................................................................................... 137
List of Tables

Table 1-1: Testing capabilities of SST and other black box security testing tools ........................................... 5
Table 2-1: The categorization of six groups of web testing approaches ......................................................... 13
Table 2-2: The top ten web applications vulnerabilities reported in 2007 .................................................. 15
Table 4-1: The categorization of markup tags .............................................................................................. 37
Table 4-2: Consequence of CIA security requirements violation .............................................................. 41
Table 4-3: Flaws caused by six markup tags. ............................................................................................... 42
Table 4-4: The format of each tag .............................................................................................................. 44
Table 4-5: Naming test cases ..................................................................................................................... 50
Table 4-6: Two mutants created by CaseSensitiveTagMutationEngine .................................................... 51
Table 4-7: Two mutants created by CharSpecificTagMutationEngine ...................................................... 51
Table 4-8: Two mutants with their dates mutated ..................................................................................... 52
Table 5-1: A total of 21 test cases generated from a nested tags packet .................................................... 80
Table 5-2: A total of 12 test cases generated according from the library .................................................. 82
Table 6-1: The constraints of the user name input string ........................................................................... 92
Table 6-2: Mutants from the case sensitive mutation engine ........................................................................ 97
Table 6-3: Two mutants created by the length mutation engine .............................................................. 98
Table 6-4: Mutants generated by the legal characters mutation engine .................................................... 99
Table 6-5: Two mutants created by the string specific mutation engine .................................................. 100
Table 8-1: Nine experiment setup summary ............................................................................................ 107
Table 8-2: Vulnerability detected by caseSensitive tag ................................................................. 108
Table 8-3: Vulnerability detected by charSpecific tag ............................................................................ 109
Table 8-4: Vulnerability detected by dateSpecific tag ............................................................................ 110
Table 8-5: Vulnerability detected by enumeratedLiteral tag ................................................................. 111
Table 8-6: Vulnerability detected by valueLimitation tag ...................................................................... 112
Table 8-7: Vulnerability detected by stringSpecific tag .......................................................................... 113
Table 8-8: Vulnerability detected by syntaxSpecific tag ....................................................................... 114
Table 8-9: Vulnerability detected by length tag ...................................................................................... 115
Table 8-10: Vulnerability detected by LegalChars tag .......................................................................... 116
Table 8-11: The second experimental setup ............................................................................................ 116
Table 8-12: Experiment two run time .................................................................................................... 117
Table 8-13: The third experimental setup information ............................................................................. 126
Table 8-14: The data from experiment three ................................................................. 126
Table 8-15: PROTOS Test-Suite: co-http-reply test cases characteristics ..................... 128
# List of Figures

- Figure 2-1: A simple web application operation ................................................................. 11
- Figure 2-2: A simple business logic of online purchasing web site ........................................ 17
- Figure 2-3: A mutant example ............................................................................................... 19
- Figure 2-4: TXL grammar and program ................................................................................. 20
- Figure 2-4a: TXL grammar ...................................................................................................... 20
- Figure 2-4b: A simple TXL program ....................................................................................... 20
- Figure 2-5: Three phases of TXL ........................................................................................... 22
- Figure 3-1: The core workflow of the SST framework .......................................................... 26
- Figure 3-2: Decomposing of five modules of the SST framework .......................................... 27
- Figure 3-3: Use case of SST .................................................................................................... 29
- Figure 4-1: The content of a manifest file ............................................................................... 33
- Figure 4-2: Enumerated non-terminal [method] .................................................................... 38
- Figure 4-3: the extra code of the modified jhead.c file ............................................................ 40
- Figure 4-4: The relationship among tags, kind of attacks and C.I.A. violations ....................... 43
- Figure 4-5: A HTTP request message before markup ......................................................... 45
- Figure 4-6: A single markup on the method POST ............................................................... 45
- Figure 4-7: Nested markups on the method POST ................................................................. 45
- Figure 4-8: A relation tag on two terminals ......................................................................... 46
- Figure 4-9: A partial multipart packet with markup tag “length” ........................................ 47
- Figure 4-10: Internal distinction between SST markup tag and XML tag ............................ 49
- Figure 4-11: Mutator generates mutants with a specific file ............................................... 52
- Figure 4-12: Enumerating all methods from the grammar .................................................... 53
- Figure 4-13: The first and the second roles of mutants ........................................................... 54
- Figure 5-1: The steps for generating test cases ..................................................................... 57
- Figure 5-2: Generalized grammar and specialized grammar .................................................. 59
- Figure 5-3: Creating a protocol description language file ....................................................... 59
- Figure 5-4: The relationships among the generalized grammar, the description file, and the protocol specification ..................................................................................... 61
- Figure 5-5: Final protocol description language file ............................................................... 62
- Figure 5-6: Generating a new grammar and rules, and an Insert Markup Program ............ 63
- Figure 5-7: The partial HTTP request grammar and the partial protocol description language .... 64
Glossary

AG – additional grammar .............................................................................................................. 90
BNF – Backus-Naur Form ............................................................................................................. 20
Bug – an informal name for failure and fault ................................................................................... 9
Capture – one of the SST module responsible for capturing packets ............................................ 26
CGI – common gateway interface .................................................................................................. 112
CIA – acronym for confidentiality, integrity, and accessibility ..................................................... 41
CSS – cascading style sheets ......................................................................................................... 15
e-commerce – electronic commerce ............................................................................................. 3
E-Crime – electronic crime .............................................................................................................. 3
Error – a mistake made by a programmer ...................................................................................... 9
Failure – a function of the program is performed incorrectly .......................................................... 9
Fault – an actual fragment of code in the program that causes failure .......................................... 9
GUI – graphical user interface ..................................................................................................... 119
HTML – hyper text markup language ........................................................................................... 12
HTTP – hyper text transfer protocol ........................................................................................... 2
ICal – iCalendar ........................................................................................................................... 119
IIS – internet information services ............................................................................................... 108
IP  – internet protocol .................................................................................................................... 7
jpeg – joint photographic experts group ....................................................................................... 39
JavaScript – a scripting language used within browsers ................................................................. 11
KDE – K Desktop Environment .................................................................................................. 119
lproto – protocol description language ........................................................................................ 48
Markup – one of the SST module responsible for inserting markup tag ........................................... 26
m-commerce – mobile commerce .................................................................................................. 3
MIME – mutipurpose internet mail extensions .............................................................................. 39
ODBC – open database connectivity ............................................................................................. 109
Oracle – one of the SST module responsible for checking testing results ..................................... 25
OWASP – the open web application security project .................................................................... 14
PDU – protocol data unit ................................................................................................................ 2
Replay – one of the SST module responsible for replaying mutated packets ............................... 26
RFC – request for comments .......................................................................................................... 10
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMA</td>
<td>response message after</td>
<td>55</td>
</tr>
<tr>
<td>RMB</td>
<td>response message before</td>
<td>55</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>SIG stands for signal and SEGV stands for segmentation violation</td>
<td>127</td>
</tr>
<tr>
<td>SOAP</td>
<td>simple object access protocol</td>
<td>25</td>
</tr>
<tr>
<td>SQL</td>
<td>structured query language</td>
<td>5</td>
</tr>
<tr>
<td>SST</td>
<td>syntax-based security testing</td>
<td>4</td>
</tr>
<tr>
<td>TCP</td>
<td>transmission control protocol</td>
<td>7</td>
</tr>
<tr>
<td>TXL</td>
<td>transformation language</td>
<td>5</td>
</tr>
<tr>
<td>URL</td>
<td>uniform resource locator</td>
<td>16</td>
</tr>
<tr>
<td>VBScript</td>
<td>visual basic scripting language</td>
<td>11</td>
</tr>
<tr>
<td>XML</td>
<td>extensible markup language</td>
<td>30</td>
</tr>
<tr>
<td>XSS</td>
<td>cross-site scripting</td>
<td>15</td>
</tr>
</tbody>
</table>
Chapter One

Introduction

Software testing is an on-going research topic in computing science. There is a plethora of literature on the testing methodologies used to test correctness and robustness of software systems. Software testing should not just focus on validation and verification [61], but also on the need to test the security of the software to prevent any unauthorized persons/machines to access the system. Despite widespread knowledge of security bug classes [47, 71], these security bugs continue to occur. Security faults cause serious consequences, such as information theft or the complete failure of the system. When comparing the security threats of stand-alone applications to network applications, the latter will face greater challenges. It is simply due to the fact that network applications, for example, online shopping sites, are widely open to the public and everyone can access them. Naturally, an effective security network communication protocol testing tool is needed.

Network protocols can be binary-based or text-based. Binary protocols are ones in which the binary representation of values are used. For example the value four may be transmitted as the binary value 0x04 (8 bits) or as the value 0x00000004 (32 bits). On the other hand, text-based protocols encode the data in ASCII or UNICODE. In this case, the value 4 might be transmitted as the binary value 0x34 (the ASCII value for 4). Binary protocols have a strict format, where the orders of data elements are given, while text protocols are more flexible. For examples, a binary packet format begins with Packet_Length, and then follows with ClientEvents.Login, String_Username and
String_Password. If we swap the positions of Packet_Length and ClientEvents.Login, then the packet will be corrupted. On the other hand, in a text-based protocol such as HTTP, MIME headers (e.g. Content-Type and Content-Length) can be in any order and the packet is still valid.

Intelligent fuzzing of binary protocols involves finding boundary values for the application. The flexibility of text protocols increases the number of tests that must be completed. There are a limited number of test cases ($2^n$) that can be obtained from binary-based protocols by fuzzing its binary value. Also there is no semantic violation by changing zero to one or vice versa. On the other hand, there are considerably more test cases that can be yielded from text-based protocols ($m^n$) and the protocol violations can be obtained by breaking the semantic and/or syntactic constraints of the protocols. Therefore, the magnitude of challenge of testing text-based protocols is much greater than the binary-based protocols.

Among the existing network security testing methodologies, mutating protocol data units (PDU) for testing communication protocols presents the most intriguing technique. This technique has been successfully used to reveal vulnerabilities in some stateless protocols [66, 69, 70] by mutating the binary based PDU. Zhang [74] adapted this method for testing state-based communication protocols. Recently, Jing et al [27] built a nondeterministic finite machine for mutation testing of protocol messages at the binary level. However, little research has been conducted on mutating text-based communication protocols such as HTTP. In this thesis, a novel robust text-based protocol testing framework is introduced. It can not only test HTTP but also all the existing text-based protocols and even any new text-based protocols implemented in the future.
1.1 Motivation and Objective

Today, electronic commerce (e-commerce) and mobile commerce (m-commerce) involve billion dollar businesses. According to an E-Crime watch survey in 2007 [10], 57% of the participants were progressively more anxious about the potential effects of e-crime. These crimes could hugely impede e-businesses growth. In order to uphold and foster e-businesses, having reliable, secure software applications is an important factor. Software testing is the cornerstone for gaining reliability and security for the software because it plays an important role in bug prevention. Therefore, robust testing techniques to test software are in obvious demand. Otherwise, if customers’ sensitive data, like credit card information, can be stolen, then there is little to no hope of the online business’ survival.

The objective of this research is to investigate an inexpensive testing technique which can effectively reveal security vulnerabilities of software which is dependent on text-based communication protocols to exchange information. An example of this are online web purchasing web sites that use HTTP text-based communication protocol to exchange information between customers and the business web server. According to Beizer “data validation is the first line of defense against a hostile world”, all input data should conform to its format and grammar and the best input format should be defined as a formal language [3]. Intuitively, the language based syntax testing method will be the best fit for testing text-based protocols.
1.2 Contribution of the thesis

In order to keep promoting e-businesses growth, secure business servers and applications are needed. However, testing processes are time consuming, tedious, and error-prone. The contribution of this thesis is to build an extensible Syntax-based Security Testing (SST) framework that is protocol independent. It can test most text-based protocols. New mutators can easily be added allowing it to test new forms of text-based attacks. In addition, the SST framework also provides an extension to allow external mutators to fuzz binary data such as mutating the uploaded picture or the attachment of a pdf file.

Testing tools should possess both high value and low cost properties. The terms high value and low cost refer to the testing effectiveness and performance, respectively. The effectiveness refers to the capability of faultfinding and the performance refers to the time required to find faults. The cost is measured by time; the testers’ time to derive test cases and the time it takes to test the program. Software developers do not need to invest any extra time to create a testing tool to perform security testing on their text-based communication protocol products. Time constraints for conducting security testing will no longer be an issue. Reliable and secure business web applications will be delivered to clients on time. As a result, customers gain confidence when buying goods online and companies increase their online businesses. Ultimately this leads to economic growth.

There is little research done on text-based security testing by using black box testing approach. Table 1-1 shows the testing capabilities of SST and other black box security testing tools to reflect the contribution of this thesis.
Table 1-1: Testing capabilities of SST and other black box security testing tools

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Protocol independent</th>
<th>SQL injection</th>
<th>Cross-site scripting</th>
<th>Session Fixation</th>
<th>Directory traversal</th>
<th>Command Injection</th>
<th>Buffer overflows</th>
<th>Denial of service</th>
<th>Bypass restriction</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Session</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[17,18,63,64,65]</td>
</tr>
<tr>
<td>WAVES</td>
<td>No</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[24]</td>
</tr>
<tr>
<td>Bypass</td>
<td>No</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>[43]</td>
</tr>
<tr>
<td>SecuBat</td>
<td>No</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>[30]</td>
</tr>
<tr>
<td>SST</td>
<td>Yes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1-1 shows four other methods that alter HTTP request messages to test web applications. The first one, User Session [17, 18, 63, 64, 65], primarily tests the functional requirements of the web application. But it can be used to test bypass restriction. The other three, WAVES [24], Bypass [43], and SecuBat [30], focus on SQL injection and cross-site attack. All four testing methods are protocol independent testing tools. However, they are only able to test HTTP protocol. On the contrary, SST not only tests most text-based protocols such as HTTP, FTP, SMTP, etc., but can also reveal all the vulnerabilities listed on Table 1-1. This thesis shows the effectiveness of the SST framework. Evaluating the performance of SST will be out of the scope of this research.

1.3 Outline of the thesis

This thesis contains a total of ten chapters and references. Chapter One is the introduction, which highlights of the robustness of SST. Chapter Two concerns background and related work. Traditional software testing and web application testing will be addressed. The popular web application vulnerabilities (for examples, SQL injection and command injection) will be listed and how these kinds of attacks work demonstrated. Mutation testing, TXL, and text-based communication protocol
applications will also be introduced in this chapter. The related work of some existing web applications security testing techniques will also be presented in this chapter. Chapter Three describes the SST testing framework overviews. Chapter Four examines the SST modules in detail. Chapter Five and Chapter Six detail the low and middle level concrete architectures and the high level concrete architectures of SST, respectively. Chapter Seven introduces the functionalities of Injector. The testing scope, a three experiment setup and the testing results discussions are in Chapter Eight. Finally, future work and conclusions are presented in Chapter Nine and Chapter Ten, respectively.
Chapter 2

Background and Related Work

In this chapter, background knowledge such as software testing overviews, web applications testing, web application vulnerabilities, mutation testing, TXL, and related work will be surveyed.

2.1 Background

There are many successful e-business web sites running HTTP on top of TCP/IP network protocols, for example, eBay.com and amazon.com. The correctness, reliability and security of their web applications are important to the success of e-business. Therefore, software testing is a serious business and testing should start from the earliest possible stage of the software life cycle.

2.1.1 Software testing overview

Software testing plays an important role in traditional software development and must be conducted thoroughly before deployment or delivery to clients. The software testing and web application testing process should consist of at least two activities: validation and verification [61]. Validation ensures that the software incorporates correct implementation (in agreement with the user requirements). Verification checks that the software has been implemented correctly (in accordance with its specifications). In other words, we want to ensure that the implemented software is what the client wants and is flawless as possible before delivery.
The process of software verification can be achieved by formal or informal testing. High security and human life related software should be tested using formal methods. However, formal testing requires extensive knowledge to model and to prove the system’s correctness. For example, the denotational semantics method [67] uses mathematical objects to represent functions and prove its correctness. This approach is not commonly used for industrial software development; it is not easily applied to the ever changing, time-to-market constrained web application software. Informal testing is widely accepted by industry. One of the reasons is that it is relatively easier to master informal testing techniques compared to formal testing methods. However, a drawback of informal testing techniques is often the tedious and confusing software testing jargon. Different testing teams may test a program with the same approach but call the testing method different names. For example, complete testing and exhaustive testing are understood to have the same meaning. We can easily find more than a hundred technical testing names. International Software Testing Qualification Board [26] has sought the views of different parties (industries, organizations, and government agencies) and attempted to standardize the glossary of terms used in software testing. Consequently, most terms in the glossary are widely accepted.

Test engineers commonly use black box and/or white box approaches to conduct testing. This is because most testing originates from these two fundamental methods. In order to achieve black box testing, there are several major methods that can be used, such as equivalence partitioning on the input and output domains, and analyzing the boundaries and extreme values. As a result, these three testing methods; equivalence partitioning testing, boundary value testing, and extreme value testing, originate from
black box testing. As for white box testing, the basic approach is to find test cases that can execute the code differently, in other words, looking at different criteria of code coverage. Therefore, white box testing yields condition testing, decision testing, path testing, and statement testing, to name a few. In short, black box and white box testing are only generic names for the software testing. The black box testing does not require the availability of the source code while conducting testing, but the white box testing does.

For the sake of consistency, four confusing words in software testing will be defined here for their use throughout this thesis.

Fault – an actual fragment of code in the program that causes failure
Failure – a function of the program is performed incorrectly
Bug – an informal name for failure and fault
Error – a mistake made by a programmer

Subsection 2.1.2 reports the contributions and efforts of numerous researchers to investigate and build different robust testing systems to conduct web applications testing.

2.1.2 Web application testing

In 1990, the first web browser Nexus was born. At that time, web application testing was not a consideration, perhaps because of simplicity and/or very limited application features. A web application is an application that is accessed via a web browser using the Hypertext Transfer Protocol (HTTP) over a network such as the Internet or intranet. To be more precise, HTTP originally is a stateless request and response protocol between clients and servers. An HTTP client (web browser) makes an initial request to an HTTP server (the web server) via Transmission Control Protocol
(TCP) establishing a connection on port 80 by default. An HTTP server listens to this port and sends the request message (application) to the HTTP client.

After two new headers, Cookie and Set-Cookie, proposed at RFC 2109 [33], web applications have become more sophisticated and complex than ever before. Many legacy software systems have also been migrated to web-based systems. Web applications tend to change more rapidly with shorter development times (time-to-market concerns) than traditional software. The short lifecycle of web application software is normally three to six months [41].

According to a study from the Business Internet Group San Francisco (BIG-SF), of forty-one web sites under US government management, twenty-eight (68.29%) contained bugs that caused web application failures [6]. These faults reside in the web documents (static pages and dynamic pages). The failures are the file not found error and incorrect data responses. Have these websites been tested before being put on the web server? As government managed websites, time-to-market constraints should be less than for commercial business websites. Does this mean that the failure rates of commercial websites are even higher? The answer to this question was revealed on November 22, 2002, the biggest shopping day of the year in the United States – the day after Thanksgiving. The Black Friday report [7] stated consumers spent $196 million online shopping. This report also stated that 72.5% of the websites suffered some kind of failure. If it were not for such a high percentage of website failures, would consumers have spent more than $196 million on this day?

Based on the fact that e-businesses create a huge economic impact, the report drew many researchers’ attention. As a result, many researchers have and continue to
propose robust testing methods to test web applications. The testing targets are primarily focused in the shaded circle shown in Figure 2-1. Within this circle, there are a total of six potential web application problems. Figure 2-1 depicts the operation of a simple web application [18].

![Figure 2-1: A simple web application operation](image)

Potential web application errors that could occur within the circle are listed below.

1. The broken static link problem – if a web page provides a link to static pages that have been removed, users will receive an error.

2. The broken dynamic link problem – when a dynamic link (or button) is used, this link triggers software possibly written in JavaScript or VBScript to perform the task. This kind of action does not require the user to input any data, unlike a form link. The problem lies with the program/software used to retrieve the data.

3. The form link problem – a form link application is often encountered by the user, an example is online purchasing. This problem is more complex than the dynamic link situation as described above. In addition to
programmer errors, users might input special characters that exercise the robustness of the application which may also cause web application failure.

4. The dynamic page creation problem – Many web applications have a dynamically created web page to reply to users’ requests. For example, a dynamic page creation may depend on the database server processing an SQL query and format the result in a page. It is worth mentioning that the server state may incur errors under some circumstances. For example, the page contains a time sensitive greeting. If a user logs on to a particular server before noon, then the dynamic created greeting page will display good morning! However, if a user logs on to the same server also before noon but at different geographic region, then a failure could occur because of the time zone difference.

5. The uncontrolled flow transaction problem – this kind of problem may cause an inconsistent web page rendering result even though the web application code is flawless. For example, after reading a given email and then using the button to return to the inbox page, the inbox shows that the email message has not been opened yet.

6. The syntax error problem – a common syntax error of the HTML code is missing a closing bracket. For example, `<h1><font color="#FF0000">How are you?</font><h1>`, if we omit the `</h1>` closing bracket, the most updated web browsers would still be able to render the “How are you?” with the text size “h1”. However, if we are missing too many closing
brackets, then the rendered page will have the anomaly situation especially if there are rendered tables.

<table>
<thead>
<tr>
<th>Models</th>
<th>Testing Methods</th>
<th>Testing criteria</th>
<th>Testing Targets</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>White Box</td>
<td>Black Box</td>
<td>Control flow</td>
<td>Data flow</td>
</tr>
<tr>
<td>Formal</td>
<td>TestUml</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Stated-based</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Object-based</td>
<td>Object-based</td>
<td>X</td>
<td>X</td>
<td>X X</td>
</tr>
<tr>
<td></td>
<td>Agent-based</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>OO Web Test</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Statistical</td>
<td>Markov Chain</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UML</td>
<td>ReWeb/TestWeb</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Slicing</td>
<td>WebApp slicing</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>User Session</td>
<td>User session data</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2-1: The categorization of six groups of web testing approaches

Although there are many potential errors that reside within the shade circle, these potential errors are all able to be found by existing web application testing methods. There are many different kinds of testing methods suggested by software engineering researchers. Table 2-1 shows the categorization of the six groups of testing approaches [31].

Each testing group approach contains the testing models, testing methods, testing criteria, and testing targets. For example, the second row of the table, the formal method group contains two testing models which are TestUML and State-based. The models represent the web application structures and methods used for test case generation. The
testing methods describe the testing mechanism; white box or black box testing. The testing criteria show the coverage requirement (data and/or control flow). The testing targets describe the functional or non-functional requirements of the software to be tested. Details can be referenced from the original papers. Researchers put a great deal of effort to explore different kinds of testing methods to ensure the web applications can fulfill the verification and validation requirements. However, these existing methods are unable to test errors that occur outside the shaded circle in the Figure 2-1. Subsection 2.1.3 introduces the most common security errors found outside the shaded area.

2.1.3 Web application vulnerabilities

Web application vulnerabilities may occur inside or outside the shaded circle. Security faults could be caused by the web server, the web applications, and/or the communication to the database server. For example, by-pass client side validation is a security fault that occurs inside the shaded circle. SQL injection is another security fault that occurs outside the shaded circle. Table 2-2 shows the top-ten web application vulnerabilities of 2007 as reported by the open web application security project (OWASP) [47].

According to OWASP reports, the top ten web application vulnerabilities are slightly different among 2003, 2004, and 2007. Although some vulnerabilities are no longer on the 2007 list, security testing should not overlook these vulnerabilities. For example, the denial of service attack is not listed in 2007. It does not mean this kind of attack no longer exists. Denial of service can be a serious fault. It can lead to ruining the reputation of a company and the loss of business. The rest of this subsection introduces the web application vulnerabilities contained in Table 2-2.


<table>
<thead>
<tr>
<th>Rank</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cross-site scripting</td>
</tr>
<tr>
<td>2</td>
<td>Injection flaws</td>
</tr>
<tr>
<td>3</td>
<td>Malicious file execution</td>
</tr>
<tr>
<td>4</td>
<td>Insecure direct object reference</td>
</tr>
<tr>
<td>5</td>
<td>Cross-site request forgery</td>
</tr>
<tr>
<td>6</td>
<td>Information leakage and improper error handling</td>
</tr>
<tr>
<td>7</td>
<td>Broken authentication and session management</td>
</tr>
<tr>
<td>8</td>
<td>Insecure cryptographic storage</td>
</tr>
<tr>
<td>9</td>
<td>Insecure communications</td>
</tr>
<tr>
<td>10</td>
<td>Failure to restrict URL access</td>
</tr>
</tbody>
</table>

Table 2-2: The top ten web applications vulnerabilities reported in 2007

**Rank 1- Cross-site scripting**

Cross-site scripting [2] (also known as XSS or CSS) is malicious code inserted into the targeted website. XSS is the more commonly used acronym because CSS is often confused with Cascading Style Sheets. If users trigger the malicious code, then their sensitive information could be stolen e.g. user id, password, and credit information. For example, there are many websites that provide guestbook or forum programs to let users submit posts containing HTML, JavaScript, and/or VBScript. If a user logs in as “X” and reads a message by “Y” that contains malicious code, then it may be possible for “Y” to hijack this user’s session and steal information by cookie theft. This theft occurs as the malicious code generates a response page after the victim’s request that will be delivered on behalf of the web application. Therefore, the victim’s browser allows the rendering of the page, thus, compromising the victim’s machine.
Rank 2 – Injection flaws

Injection flaws include SQL injection and command injection. SQL injection attack is specially crafted malicious patterns of structured query language or system commands [1, 8, 9, 13, 20, 21, 62] sent to the web application server. Hackers can exploit a web application system by launching this kind of attack. An example of SQL injection is shown below.

```
SELECT * FROM Users WHERE (UserID = 'A' OR 'B' = 'B') AND (Password = 'A' OR 'B' = 'B').
```

The WHERE clause will have no real effect because the predicate ‘B’ = ‘B’ will always be evaluated as TRUE. Attackers will successfully bypass the web application’s authentication if the web application does not have a filtering mechanism e.g. Scott and Sharp [59] filter tainted input data.

Command injection attack utilizes operating system commands to steal information. Attackers piggyback system commands by using the semicolon character or a new line character while performing a form submission. The operating system interprets these characters as the end of one command and the start of a new command.

Rank 3 – Malicious file execution

Malicious file executions allow attackers to upload a hostile file to the server. This hostile file may be accessed by the attackers themselves or other users at a later time. As a result, the malicious code will be executed to compromise the system.

Rank 4 – Insecure direct object reference

Insecure direct object reference is also referred to as, directory traversal or URL jumping. For most Linux operating systems, users’ passwords are stored in a file called
shadow and located in /etc directory. If a specially crafted URL like http://www.example.com/page1.php?a=../../../etc/shadow successfully traverses to this directory, then attacks will gain all the users’ passwords. As for URL jumping, Figure 2-2 depicts the simple business logic of online purchasing web site.

![Figure 2-2: A simple business logic of online purchasing web site](image)

If attackers are able to jump from a logon page to a shipping info page, then attackers will receive their items free.

**Rank 5 – Cross site request forgery**

Cross site request forgery is not exactly like XSS; however, the nature of the attack is quite similar. The attacker builds a web site and specially crafts an image element `<img src="https://bank.ca/withdraw?account=victim&amount=100&to=attacker">` on the image link. The attacker then lures his prey to click on this link. If the victim’s banking session cookie has not yet expired, this cross site request will be launched successfully.

**Rank 6 – information leakage and improper error handling**

Sometimes applications can unintentionally leak information about the system configuration (too much detail response when failure occurs) and internal structure of the system (developers may forget to remove any comments). Attackers take advantage of this to prepare their serious attacks.
**Rank 7 – Broken authentication and session management**

Broken authentication and session management attack could allow attackers to impersonate other users to steal information. Web authentication using userid and password is a common practice. Weak credential management functions such as password change, forgot my password, and remember my password could open up a doorway for potential attacks. As for session management, a user’s session ID should be protected by secure sockets layer (SSL).

**Rank 8 – Insecure cryptographic storage**

The most common flaw is to store unencrypted sensitive data, for example, password and credit data or the use of weak algorithms for encryption.

**Rank 9 – Insecure communication**

Insecure communication can be easily handled by using SSL to send sensitive data back and forth. This kind of vulnerability should not occur.

**Rank 10 – Failure to restrict URL access**

Failure to restrict URL access attack can cause illegal access information or use functions. Although some links are not provided for unauthorized users to access, attackers are able to make the links to visit these pages or use functions. Access control checks should be conducted before allowing users to access any sensitive pages or use functions.
2.1.4 Mutation testing

Mutation testing [44, 45, 46] measures the effectiveness of a given testing method or test suite. We can generate variants of a piece of source code with small changes. Each variant is called a mutant and each mutant differs from the original in some way. However, some mutants may have the same functionality as the original program; these are called equivalent mutants. A mutant example is shown in Figure 2-3. If the testing method or test suite can reveal the non-equivalent mutants (called mutant-kill), it reflects that the testing method or test suite is able to find faults. Then, we can measure effectiveness of the testing method by the number of mutants that are killed. Ideally, we want the testing method or the test suite to kill all the non-equivalent mutants.

<table>
<thead>
<tr>
<th>Original code with no faults:</th>
<th>Mutation code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>..................</td>
<td>...........................................</td>
</tr>
<tr>
<td>..................</td>
<td>...........................................</td>
</tr>
<tr>
<td>if (A &lt;= B)</td>
<td>if (A &lt; B)</td>
</tr>
<tr>
<td>A = true;</td>
<td>A = true;</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>B = true;</td>
<td>B = true;</td>
</tr>
<tr>
<td>..................</td>
<td>...........................................</td>
</tr>
<tr>
<td>..................</td>
<td>...........................................</td>
</tr>
</tbody>
</table>

Figure 2-3: A mutant example

Mutation testing techniques have also successfully been applied to test network protocols [66, 69, 70, 74]. However, mutants are not obtained from a source code like the mutant shown in Figure 2-3. Mutants are the variants of protocol data units (PDUs). Each variant of PDU is just like the variant of source code. Changing a line of code from if (A <= B) to if (A < B) has changed the semantic of the predicate statement. Changing a line of code from if (A <= B) to if (A ; B) breaks the syntactic grammar of the programming language. Similarly, PDU has a specific grammar of its own right. Fuzzing data is not
merely changing the value of the data. For example, a HTTP packet contains GET / received.html HTTP/1.1\r\n. Changing the method GET to DELETE changes the semantics of the method. Changing the method GET to get breaks the syntactic grammar of HTTP. SST uses this testing technique for security testing on text-based communication protocols.

2.1.5 TXL source transformation

TXL [14] is a programming language designed to support transformational programming. The basic paradigm of TXL is to transform the input source to a desired output in accordance with a set of transformation rules. TXL program basically contains two parts. The first part of a TXL program defines a context free grammar of an input source. The structure of the grammar is similar to Extended Backus-Nauer Form (BNF). Figure 2-4a shows a very simple TXL grammar to define first name and last name.

| define program [name] end define |
| define name [firstName][lastName] end define |
| define firstName [id] end define |
| define lastName [id] end define |

**Figure 2-4a: TXL grammar**

| function main replace [program] P [program] construct newName [id] ‘Wayne by P [changeFirstName newName] end function |
| function changeFirstName newName [id] replace [program] P [program] deconstruct P fName [id] lName[id] by newName lName end function |

**Figure 2-4b: A simple TXL program**

Figure 2-4: TXL grammar and program
The second part defines a set of transformation rules to transform the input source to the desired output and the main program. Every TXL program must have a main program. Figure 2-4b illustrates a simple transformation rule `changeFirstName` to change the input first name to a desired `newName` and the main program `main`. Once the TXL program is executed, it goes through three phases. The first phase is the parsing phase. The TXL robust parser tokenizes the entire input source and generates a parse tree according to the grammar of the input source which is defined at the first part of the TXL program. For this example, there are two tokens which are `David` and `Palmer`. Both tokens are the predefined non-terminal `id` type. The second phase is the transformation. Program execution starts from the entry point (function `main`) and sets the entire input tree as a scope of the tree for transformation. After creating a new variable named `newName` with the value `Wayne` by a constructor, the program then invokes the transformation function `changeFirstName` with the `newName` as a parameter to perform the transformation. The transformation rule deconstructs the tree and yields the first name `David` and the last name `Palmer`. Then, the `firstName` `David` is replaced by the `newName` `Wayne` and a new desired tree is formed. The final phase, TXL simply un-parses the new tree and produces the output text. Figure 2-5 shows the three phases of TXL execution.
One of the robustness features of TXL is support agile parsing [16]. The basic idea of agile parsing is overriding non-terminals. It allows the programmer to explicitly redefine a non-terminal with the key word `redefine` to a new definition more appropriate to a special task. It can speed up the SST to create mutants. Please refer Chapter 4 for details.

### 2.2 Related work

Web application security testing is not a new research topic. Security threats can be identified by white box testing (static analysis and/or dynamic analysis), or black box testing or both. According to Pistoia et al’s survey [48], there are many security testing practitioners that use a white box approach but few have been reported to utilize the black box approach. Additional white box testing approaches can be found in papers such as [23, 25, 38].
Elbaum et al introduce an inexpensive User Session Data method [17, 18] for web application testing. The test cases for the user session based testing can be obtained on-the-fly with minimal effort from the instrumented web server. The instrumented web server logs user’s URL and name-value into a log file, then applies heuristics to generate test cases. Elbaum et al suggest three approaches of replaying user sessions. The first approach is to replay an individual user’s sessions sequentially. The second approach is to replay different combinations of users’ sessions. The third approach is to reuse user sessions with Form modifications. Although Sprenkle et al [63, 64, 65] found that Elbaum et al’s mixing of different user session test cases lost the application state dependencies; both test case generation methods for user session testing can be used as a black box testing approach to test the state of the web applications.

Huang et al [24] propose very expensive mechanisms for assessing Web application security named WAVES. (WAVES was an open source project at http://waves.sourceforge.net but is no longer available.) WAVES uses a black box approach for security testing. It can detect web application security vulnerabilities such as SQL injection and cross-site scripting (XSS). WAVES consists of a crawler to crawl the web application in order to find all pages in the website that contain HTML forms. All the HTML forms are data entry points. WAVES uses algorithms to send malicious SQL to the server side program via the HTML forms.

In order to detect cross-site scripting attacks, WAVES launches an initial run to trigger a learning mode to crawl through all predefined links that are unlikely to contain malicious code. During the crawling process, the normal behavior of the web application is studied and recorded into a hashed policy database. This policy database will be used
to find any suspected code insertion to the web application. A shortcoming of WAVES for detecting XSS is that it only finds malicious code inserted into the web application. It does not find whether the web application is vulnerable to attack by XSS.

Kals et al [30] also use a black box approach for web application security testing. Their testing tool named SecuBat consists of three main components (crawler, attacker, and analyzer). The concept of crawling pages and launching attacks are similar to that of WAVES. However, the XSS detection between SecuBat and WAVES are different. SecuBat focuses on finding any scripting languages embedded in the response page, while WAVES compares differences during page execution with the policy database. WAVES and SecuBat only detect SQL injection and XSS attacks. Neither of them can detect other attacks like buffer overflows. In addition, both of their testing methods have false positive problems.

Bypass testing, proposed by Offutt et al [43], is another black box approach for web application security testing. The test cases in Bypass testing skip or bypass the client-side input validation, therefore going directly to the server-side program. Test cases may contain violations of constraints among different parameters and between software components. Violations can occur at the value level, parameter level, and/or control flow level.

None of the above existing black box approaches for web application security testing is similar to this research testing method. The significant differences are outlined below –

• The existing methods only test the web applications themselves. This research approach is a ‘kill two birds with one stone’ testing method. We not only test the
security aspect of the web applications, but also the robustness of the server side configurations. For example, some versions of web servers may not be able to handle certain kinds of attacks, like HTTP smuggling [47] but the web applications themselves have no securities issues.

- Some existing methods replay the captured users input data as test cases. Others use algorithms to auto fill the user input forms as test cases. Our test case generation will be more sophisticated. First, our group has modified the Firefox web browser to capture the HTTP request and response packets. Second, we have based grammar of the communication protocols which include the low level protocol HTTP request and response, the middle level protocol SOAP, and the high level application protocol like shopping cart, as protocol specifications. Then we use these protocol specifications to generate testing cases. Large amounts of test cases can be generated easily by using mutation technique to break the constraints of the protocol specifications. In addition, one of the robustness testing tactics of the server using a user single HTTP request sent (a sequence of request and response message passing between client and server are captured), would require rearranging the sequence of the request/response order randomly and rerouting back to the server to perform robustness testing.

- The modified Firefox browser also captures the HTTP response and will be used as an oracle for our testing.
Chapter 3

Syntax-based Security Testing Overview

SST testing framework is based on the previous research of our Protocol Tester group [66, 69, 70, 74]. However, this research focuses on text-based low, middle, and high level of security testing. Figure 3-1 illustrates the core workflow of the SST framework.

![Figure 3-1: The core workflow of the SST framework](image)

In Figure 3-1, the SST framework consists of a total of five modules; Capture, Markup, Mutate, Replay, and Oracle. The Capture module is a protocol dependent module and the other four are protocol independent. The Capture module prepares test data. The Markup module is responsible for marking up variables as targets for mutation. The Mutate module performs mutations. The Replay module sends mutated packets to the targeted server. The Oracle module draws a conclusion of the testing results. Chapter
5 and Chapter 6 thoroughly discuss low/mid level protocol specification and high level protocol specification, respectively.

Figure 3-2 further decomposes the five modules of SST and using HTTP as an example. The protocol dependent module Capture contains Sniffer and Decoder. The protocol independent modules are Markup (containing Protocol Specification, Markup Tags, and Tags dictionary), Mutate (containing Mutators), Replay (containing Injector, RealTimeUpdate, and Travel Agent), and Oracle. The concrete architectures of these five modules were designed from scratch in order to conduct the syntax-based security testing for text-based communication protocols.
Among these five modules, there are nine submodules, the Sniffer and the Decoder are protocol dependent submodules, and the other seven submodules are protocol independent (Protocol Specification, Markup Tags, Tags dictionary, Mutators, Injector, RealTimeUpdate, Travel Agent). Figure 3-3 shows all functional requirements of SST. Functions that originate from the protocol dependent modules will be depicted by a thick black line. The activities involved in SST are presented below.

1. The Sniffer captures the HTTP request and response packets, and generates a manifest file.

2. The HTTP response packets will then be decoded by the decoder if the response message is not in plain text format.

3. Creating a protocol specification.

4. The HTTP request packets are then inserted with markup tags.

5. Mutator mutating the packets in accordance with the markup tags.

6. Checking whether or not the replayed packet is necessary to perform system state update and database state restoration.

7. Collecting the targeted server network information, type of text transfer network protocol, port number, and/or via proxy server to be used from the manifest file.

8. Replay this packet and decode the response messages if necessary.

9. At length, the oracle compares the original response package with the replayed response package to verdict the testing result.
3.1 Protocol dependent module

The Capture module contains the Sniffer and the Decoder protocol dependent submodules. The Sniffer is a modified Firefox browser. It provides three functions. The first two are capturing request packets and response packets. In the case that the captured response packets are compressed (some web servers compress the response messages), a
decoder will be invoked to decode the packets and change them back to plain text format. The third function is to create a manifest file. The manifest file contains targeted servers IP addresses, port numbers and the information of proxy servers. Therefore SST can test multiple server systems.

3.2 Protocol independent modules

The protocol independent modules are Markup, Mutate, Replay, and Oracle. The Markup module contains protocol specification, markup tags, and tag dictionary submodules. A request packet is an instance of protocol specific. Markup tags will be inserted onto the request packet by programs. Inside each markup tag is a mutation target. The markup tags are used to specify the method to mutate the content. The Mutate module contains mutators. The mutator mutates the request packet in accordance with the tag dictionary and the markup tags. The tag dictionary contains all markup tag names belonging to SST; therefore, the mutation engine can avoid confusion if the request packet contains HTML or XML. The Replay module contains Injector, RealTimeUpdate and Travel Agent submodules. The Injector replays the mutated packet, and ensures whether or not to perform a real time update and/or restore the database. The current implementation of RealTimeUpdate is HTTP specific but can be made protocol independent by adopting the approach of Zhang et al [74]. The Travel Agent sends the mutated packet to the targeted server and receives the response messages. The restore database restores the original database state after each test run. Lastly, the Oracle is responsible for checking the testing results. A sequence diagram in Figure 3-4 illustrates the workflow of SST.
Figure 3-4: The workflow of SST

1: start
2: request
3: response
4: decode
5: store original response
6: parsing request packet
7: insert tags
8: start mutation
9: check tag is valid
10: create test cases
11: line up test cases
12: check system state
13: update system state
14: check database state
15: maintain database state
16: open network sockets
17: replay modified request message
18: response
19: check message is compressed
20: call decoder
21: decompress message
22: store replayed response message
23: compare with the original response message
24: compare with the replayed response message
25: testing report
Chapter 4

Syntax-based Security Testing Module Anatomy

4.1 Network sniffer

The network sniffer is responsible for preparing all request messages for mutation and all response messages for the oracle. One original request could have a sequence of request and response messages. The request messages are stored in a request message directory, ready for the test cases generator to consume. The response messages are stored in a response message directory named Response-Message-Before (RMB) for the oracle. If the response message is compressed, then it will decompress the message before storing it to RMB. Similarly, if the response message is chunked, then it will merge the messages before storing it to RMB. This can be done using a network sniffer like Wireshark [73] or Snort [60]. However, it will fail to capture the request messages in plain text format if it has been sent over SSL. Therefore, we used a modified version of the open source web browser Firefox [42] to capture the request and response message sequences.

4.2 Manifest file

The modified Firefox browser also generates a manifest file. Figure 4-1 shows the content of a manifest file. There are three rows in Figure 4-1. Each row represents a particular packet sent manifest. For example, the third row in Figure 4-1, the 3 means the third packet of the request message sequence use https protocol via port 443 to send a request message to the host 192.168.0.105. usingProxy and proxyInfo provide proxy information if proxy server is in use.
4.3 Protocol specification

SST use protocol specifications to generate test cases. Protocol specifications are created based on the syntax specifications of each level of protocol (low, middle and high). These protocol specifications capture the instances of the classes of constraints for a particular testing protocol. Test cases can be obtained by using transformation rules to break its constraints. The protocol specification can also be in compositional form for other protocol specifications. For example, if we have a protocol specification for the low level HTTP protocol, then we can extend this protocol specification to become a middle level XML SOAP protocol specification. Again, we can also extend the protocol specification to a high level shopping cart protocol specification. TXL grammars will be used to represent this protocol specification. The partial protocol specifications of the low level HTTP protocol and the middle level SOAP protocol are shown below.

Example One: the partial low level HTTP protocol specification

% partial HTTP grammar
define program
    [request-message]
end define

define request-message
    [request-line][repeat headers_message][CRLF][opt message_body]
end define

define request-line
    [method][space][request-uri][space][http-version][CRLF]
end define
define method
    OPTIONS | GET | HEAD | POST | PUT | DELETE | TRACE | CONNECT
end define

define headers_message
    [general_header][CRLF]
    | [request_header][CRLF]
    | [entity_header][CRLF]
end define

define entity_header
    [Allow]
    | [Content_Encoding]
    | [Content_Language]
    | [Content_Length]
    | [Content_Location]
    | [Content_MD5]
    | [Content_Range]
    | [Content_Type]
    | [Expires]
    | [Last_Modified]
    | [extension_header]
end define

define message_body
    [repeat token_or_key]
end define

The repeat token_or_key idiom used in message body is used to parse all tokens until the end of the file as the message body. Markup tags can be inserted by surrounding any non-terminals of the protocol specification e.g. <enumeratedLiteral>method</enumeratedLiteral>. After the enumeratedLiteral tag is inserted surrounding the non-terminal method of the HTTP.grm, the protocol specification not only retains the original grammar specification but also adds semantics to the grammar. For the above example, the original method can only be one of the following methods; OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, and CONNECT. However, the enumerated tag allows the method to be any one of them. The second example shows how to extend
the low level protocol specification to become a middle level XML SOAP protocol specification.

**Example Two: the middle level XML SOAP protocol specification**

In order to extend the low level protocol specification to the middle level protocol specification, we can just simply use a redefine statement to extend the low level HTTP protocol specification to the middle level XML SOAP protocol specification. This partial grammar example is as follows –

```plaintext
include "http.grm"

redefine entity_header
  ...
  | [SOAPAction]
end redefine

define SOAPAction
  [soap_uri][soap_message]
end define

define soap_message
  [xml_declaration] [open_soap_envelope] [soap_header] [soap_body]
  [close_soap_envelope]
end define
...
```

As for the high level application model, we can also use the redefine statement to redefine the message-body to add a shopping_cart non-terminal into either the low level HTTP protocol specification or the middle XML SOAP protocol specification to yield the high level application protocol specification. Consider a very basic online shopping web application; users are typically required to fill out some forms (personal particulars and/or selected items) then click the submit button.
A request message with a simple form

POST /received.html HTTP/1.1\r\nHost: 192.168.0.2:8080\r\nUser-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220
BonEcho/2.0.0.11\r\nAccept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8
image/png,*/*;q=0.5\r\nAccept-Language: en-us,en;q=0.5\r\nAccept-Encoding: gzip,deflate\r\nAccept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r
Keep-Alive: 300\r\nConnection: keep-alive\r\nReferer: http://192.168.0.2:8080/\r\nContent-Type: application/x-www-form-urlencoded\r\nContent-Length: 19\r\n\r\nfname=Ben&lname=Kam\r\n
We can extract the variables name-value pairs from this request message. The target information will be exactly 19 bytes (Content-Length: 19) that can be found in the message body. However, in order to build the shopping cart protocol model, some idea about the web application structure must be known so that a relationship between the variables and the information they represent can be deduced. From the above request message, `fname` and `lname` are responsible for the user’s first name and last name, respectively. Once we identify all the variables, then we can build the high level protocol specification.

Example Three: the high level shopping cart protocol specification

include "http.grm"

redefine message-body
  [shopping_cart_message]
end redefine

define shopping_cart_message
  [fname] [lname]
end define
Of course, building a high level protocol specification will be based on the online purchasing web application to reflect each individual case (vendor). Humans or design recovery tools must be involved in this process.

### 4.4 Markup tags

<table>
<thead>
<tr>
<th>Types</th>
<th>Tags</th>
<th>Apply to</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntactic</td>
<td>enumeratedLiteral</td>
<td>Low/mid level</td>
<td>Change to another terminal provided from grammar to alter the original semantics</td>
</tr>
<tr>
<td></td>
<td>caseSensitive</td>
<td>Low/mid level</td>
<td>Change the terminal letters from upper case to lower case or vice versa so as to violate the syntax constraint</td>
</tr>
<tr>
<td></td>
<td>CaseSensitivie</td>
<td>High level</td>
<td>Change the terminal letters from upper case to lower case or vice versa so as to violate the syntax constraint</td>
</tr>
<tr>
<td></td>
<td>charSpecific</td>
<td>Low/mid level</td>
<td>Change the terminal character so as to violate the syntax constraint</td>
</tr>
<tr>
<td></td>
<td>dateSpecific</td>
<td>Low/mid level</td>
<td>Change the terminal date format so as to violate the syntax constraint</td>
</tr>
<tr>
<td></td>
<td>syntaxSpecific</td>
<td>Low/mid level</td>
<td>Alter the terminal characters so as to violate the syntax constraint</td>
</tr>
<tr>
<td></td>
<td>valueLimitation</td>
<td>Low/mid level</td>
<td>Change the terminal value so as to create buffer overflow</td>
</tr>
<tr>
<td></td>
<td>Length</td>
<td>High level</td>
<td>Violate the maximum and minimum limitation of the input string</td>
</tr>
<tr>
<td></td>
<td>LegalChars</td>
<td>High level</td>
<td>Insert some illegal characters into the input field so as to violate the syntax constraint</td>
</tr>
<tr>
<td>Relational</td>
<td></td>
<td>Low/mid level</td>
<td>Break the semantics relationship of any dependent terminals</td>
</tr>
<tr>
<td>Custom</td>
<td>length</td>
<td>High Level</td>
<td>Insert a specially crafted string into the input field so as to launch a special kind of attack. For example SQL injection</td>
</tr>
<tr>
<td>External</td>
<td>Jpeg</td>
<td>High Level</td>
<td>Invoke external Jpeg mutator to fuzz the jpeg file.</td>
</tr>
</tbody>
</table>

Table 4-1: The categorization of markup tags

SST provides four types of markup tags. First, the syntactic type of tag enumerates the non-terminal options from the grammar. Second, the lexical type changes the lexicon of the terminal. Third, the relational type breaks the semantics relationship
between terminals. Last, the custom type places a specially crafted string into the application level input form field. Table 4-1 shows the categorization of the markup tags.

**Syntactic type**

Syntactic type markup tag analyzes the marked up grammar and generates alternative grammar automatically. The enumeratedLiteral is a syntactic type markup tag.

```
define request-line
  <enumeratedLiteral>[method]</enumeratedLiteral>[space][request-uri][space]
  [http-version][CRLF]
end define

define method
  OPTIONS | GET | HEAD | POST | PUT | DELETE | TRACE | CONNECT
end define
```

Figure 4-2: Enumerated non-terminal [method]

For example, the non-terminal method in Figure 4-2 is marked up by the enumeratedLiteral. Assume the instance of this protocol specification contains the method POST, then the enumeratedLiteral mutator will enumerate the other alternative methods (OPTIONS, GET, HEAD, PUT, DELETE, TRACE, CONNECT) based on the grammar to generate mutants. Please refer to Section 5.5 for detail.

**Lexical type**

The Lexical type markup tag violates the protocol specification syntax constraint.

There are a total of eight different lexical type markup tags shown in Table 4-1. caseSensitive tag changes the terminal letters from upper case to lower case or vice versa so as to violate the syntax constraint. The purpose of CaseSensitive tag is the same as caseSensitive tag however CaseSensitive is used at the high level protocol specification.
Relational type

The Relational markup tag breaks the semantics relationship of any dependent terminals. The length is a relational markup tag. For example, the Content-Length of HTTP MIME header is 48, it means that the message body contains exactly 48 characters. In order to violate this relationship, the length tag has to be placed at both the Content-Length and the message_body. Please refer to Figure 4-8 for detail.

Custom type

The Custom markup tag inserts a specially crafted string into the input field so as to launch a certain kind of attack e.g. SQL injection. The stringSpecific is a custom markup tag and is used at the high level protocol specification.

External type

The SST framework provides an extension to invoke external mutators to perform mutation. This extension makes the SST framework perfect. The SST framework no longer only mutates text-based files, but also binary-based files. In some situations, text-based communication protocol involves binary data. For example, uploading jpeg files or pdf files to the web server are very common HTTP web applications. In this research, an external mutator for fuzzing jpeg images is chosen for demonstration. The Jpeg tag is an external markup tag. It invokes the external mutator jhead to perform mutation. The jhead is an open source (Linux package) which provides functions to modify the Exif data of the jpeg file. In order to have a serious violation of the jpeg file format, the jhead version 2.87 open source has been modified so that it allows arbitrary changes of the width and the height of the jpeg file. For example, if we want to change the width and the height of
the image.jpg to -1 and 0 respectively, then we can execute the jhead with a flag –wh along with parameters -1 and 0 (./jhead –wh -1 0 image.jpg). Figure 4-3 shows all the extra code added to the jhead.c.

```c
//add the following lines after the line 87 of the original file
int SizeMutate = FALSE
    short width = 0, height = 0;
//add the following lines after the line 809 of the original file
if (SizeMutate){
    if(!ReadJpegFile(FileName, READ_METADATA | READ_IMAGE)) return;
    mutateDims(width, height);
    WriteJpegFile(FileName);
    return;
}
//add the following lines after the line 1383 of the original file
void setDims(uchar * Data, short width, short height){
    Data[3] = (uchar)(0xFF & (height >> 8));
    Data[4] = (uchar)(height);
    Data[5] = (uchar)(width >> 8);
    Data[6] = (uchar)width;
}
Void mutateDims(short width, short height){
    int a;
    int SectionsRead = getSectionsRead();
    Section_t *ps = getSections();
    for(a=0; a<SectionsRead; a++){
        switch(ps[a].Type){
            case M_SOF0:
            case M_SOF1:
            case M_SOF2:
            case M_SOF3:
            case M_SOF5:
            case M_SOF6:
            case M_SOF7:
            case M_SOF9:
            case M_SOF10:
            case M_SOF11:
            case M_SOF13:
            case M_SOF14:
            case M_SOF15:
                setDims(ps[a].Data, width, height);
                default:;
        }
    }
}
//add the following lines after the line 1437 of the original file
}else if (!strcmp(arg, "-wh")){
    width = atoi(argv[++argn]);
    height = atoi(argv[++argn]);
    SizeMutate = TRUE;
}
```

Figure 4-3: the extra code of the modified jhead.c file
The markup tags allow testers to plan the specific mutation of the packets in an attempt to achieve a particular abnormal behaviour of the system by processing this mutated packet during run time. For example, the intended use of the valueLimitation tag is to cause buffer overflow in the software being tested. The mutated packet is slightly different than the original, based on the type of the markup tag(s) for alteration.

There are many security flaws that can be found in literature about web applications security testing. These flaws are created by violating the fundamental CIA security requirements. CIA stands for confidentiality, integrity, and availability, respectively. Confidentiality defines only authorized users to have a right to access particular data. Integrity ensures no data can be alternated by an unauthorized user. Availability provides data that should always be available for all legitimate users.

CIA security requirements violation by web applications can cause the following consequences. First of all, it is possible to cause the web application, database, and/or web server to crash. Second, users’ data and/or system information can be stolen and/or modified. Third, computer resources are wasted by illegal users. Table 4-2 shows different kinds of security flaws caused by breaking CIA security requirements.

<table>
<thead>
<tr>
<th>CIA security requirements violation</th>
<th>Security flaws</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>Information stolen</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Information alternation</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Privacy violations</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Impersonation</td>
</tr>
<tr>
<td>Integrity</td>
<td>Web application crash</td>
</tr>
<tr>
<td>Integrity</td>
<td>Web server crash</td>
</tr>
<tr>
<td>Integrity</td>
<td>Database crash</td>
</tr>
<tr>
<td>Integrity</td>
<td>Information alternation</td>
</tr>
<tr>
<td>Availability</td>
<td>Wasting computer resources</td>
</tr>
<tr>
<td>Availability</td>
<td>Take over the system</td>
</tr>
</tbody>
</table>

Table 4-2: Consequence of CIA security requirements violation
A slight change in the content of the packet by breaking the syntax and/or semantics of the grammar will break the CIA. SST provides markup tags to instruct mutation engines explicitly to perform the changes. Table 4-3 denotes the markup tags that can possibly launch certain kinds of attacks against web applications.

<table>
<thead>
<tr>
<th>Markup Tags</th>
<th>Attacking Targets</th>
<th>Revealing vulnerability</th>
<th>Consequences</th>
<th>CIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;caseSensitive&gt;</td>
<td>User input data</td>
<td>Bypass Restriction</td>
<td>Web application and/or database crash</td>
<td>A</td>
</tr>
<tr>
<td>&lt;charSpecific&gt;</td>
<td>User input data</td>
<td>SQL Injection</td>
<td>Information stolen or alternated</td>
<td>CI</td>
</tr>
<tr>
<td>&lt;dateSpecific&gt;</td>
<td>User input data</td>
<td>Buffer overflows</td>
<td>Web application and/or database crash</td>
<td>A</td>
</tr>
<tr>
<td>&lt;valueLimitation&gt;</td>
<td>User input data</td>
<td>Buffer overflows</td>
<td>Web application and/or database crash</td>
<td>A</td>
</tr>
<tr>
<td>&lt;stringSpecific&gt;</td>
<td>User input data</td>
<td>Cross-site scripting</td>
<td>Fraudulent web site content, Information stolen</td>
<td>CI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL injection</td>
<td>Information stolen or alternated</td>
<td>CI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Directory Traversal</td>
<td>Privacy violations</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Session Fixation</td>
<td>Impersonation</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Denial of Service</td>
<td>Wasting computer resources</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command Injection</td>
<td>Take over the system</td>
<td>CIA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bypass Restriction</td>
<td>Web application and/or database crash</td>
<td>A</td>
</tr>
<tr>
<td>&lt;Legalchars&gt;</td>
<td>User input data</td>
<td>Bypass Restriction</td>
<td>Web application and/or database crash</td>
<td>A</td>
</tr>
</tbody>
</table>

Table 4-3: Flaws caused by six markup tags.
Note that enumeratedLiteral tag is not included in Table 4-3. In this particular case, the enumeratedLiteral tag is used to test the web configurations. The lattice in Figure 4-4 illustrates the relationship among tags, kind of attacks and breaking CIA.

![Figure 4-4: The relationship among tags, kind of attacks and C.I.A. violations.](image)

Assume a tester wants to test whether or not the web application will leak any confidential data. Using the above lattice, the tester can follow the attribute Confidentiality to locate which tag(s) can launch an attack to break the confidentiality security requirement.
Markup tag is an XML tag for marking up desired non-terminals of a given grammar. SST provides a total of six markup tags for low and/or middle levels protocol specification which are caseSensitive, charSpecific, dateSpecific, enumeratedLiteral, syntaxSpecific and valueLimitation tags. As for a high level protocol specification, there is a relation tag which is length, an external Jpeg tag and a stringSpecific tag. Markup tags for a high level protocol specification will be discussed in detail in Chapter 6. Each tag has an open tag and a close tag. Table 4-4 illustrates the format of six tags.

<table>
<thead>
<tr>
<th>Tag name</th>
<th>Open tag</th>
<th>Close tag</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 caseSensitive</td>
<td>&lt;caseSensitive&gt;</td>
<td>&lt;/caseSensitive&gt;</td>
<td>&lt;caseSensitive&gt;GET&lt;/caseSensitive&gt;</td>
</tr>
<tr>
<td>2 charSpecific</td>
<td>&lt;charSpecific&gt;</td>
<td>&lt;/charSpecific&gt;</td>
<td>&lt;charSpecific&gt;;&lt;/charSpecific&gt;</td>
</tr>
<tr>
<td>3 dateSpecific</td>
<td>&lt;dateSpecific&gt;</td>
<td>&lt;/dateSpecific&gt;</td>
<td>&lt;dateSpecific&gt;10/30/2009&lt;/dateSpecific&gt;</td>
</tr>
<tr>
<td>4 enumeratedLiteral</td>
<td>&lt;enumeratedLiteral&gt;</td>
<td>&lt;/enumeratedLiteral&gt;</td>
<td>&lt;enumeratedLiteral&gt;GET&lt;/enumeratedLiteral&gt;</td>
</tr>
<tr>
<td>5 syntaxSpecific</td>
<td>&lt;syntaxSpecific&gt;</td>
<td>&lt;/syntaxSpecific&gt;</td>
<td>&lt;syntaxSpecific&gt;Host&lt;/syntaxSpecific&gt;</td>
</tr>
<tr>
<td>6 valueLimitation</td>
<td>&lt;valueLimitation&gt;</td>
<td>&lt;/valueLimitation&gt;</td>
<td>&lt;valueLimitation&gt;10&lt;/valueLimitation&gt;</td>
</tr>
</tbody>
</table>

Table 4-4: The format of each tag

All markup tags can be used for a single markup or nested markups. Each tag will associate with a specific mutator. The mutator will get the content which is within the tag to perform alteration. Please refer to Section 4.8 Mutation engine for detail. Figure 4-5 shows a HTTP request message before markup. Figure 4-6 and Figure 4-7 show a single markup and nested markups done on the HTTP request message, respectively. The process of inserting markup tags on to the packet is completely automatic. Please refer to Sections 5.1 through 5.3 for detail.

The nested tags structure shown in Figure 4-7 is in well-formed order. There are no limits as to how many tags can be nested together but the start tag and the end tag of each individual tag cannot be crossed.
Figure 4-5: A HTTP request message before markup

```
POST /return.asp HTTP/1.1
Host: 192.168.1.105
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.1.105/
Content-Type: application/x-www-form-urlencoded
Content-Length: 48
```

Figure 4-6: A single markup on the method POST

```
<enumeratedLiteral><enumeratedLiteral>POST</enumeratedLiteral>/return.asp HTTP/1.1
Host: 192.168.1.105
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.1.105/
Content-Type: application/x-www-form-urlencoded
Content-Length: 48
```

Figure 4-7: Nested markups on the method POST

```
<enumeratedLiteral><caseSensitive>POST</caseSensitive>/return.asp HTTP/1.1
Host: 192.168.1.105
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.1.105/
Content-Type: application/x-www-form-urlencoded
Content-Length: 48
```
Lastly, the relation class of tag (length) is used to link together two or more terminals that share some semantic relationship. For example, the value of the MIME header Content-Length in Figure 4-7 is 48. This means that the message body should have a total of 48 bytes. Figure 4-8 shows a relation tag on these two terminals.

```
POST /return.asp HTTP/1.1
Host: 192.168.1.105
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.1.105/
Content-Type: application/x-www-form-urlencoded
Content-Length: <length id="1" root="request_message" role="length">48</length>

<length id="1" root="request_message" role="value">FirstName=John&LastName=Smith&DOB=10%2F15%2F1980</length>
```

Figure 4-8: A relation tag on two terminals

Note that the length tag has three parameters. The first one is \texttt{id}. \texttt{id} is a variable identifier and its value is a unique number which specifies which non-terminals are related. In this example, the value of Content-Length is related to the message body (48 is related to \texttt{FirstName=John&LastName=Smith&DOB=10\%2F15\%2F1980}). It is because both of them have a length tag with \texttt{id=“1”}. If there are more than one set of relation tags, then the \texttt{id=“2”} will link the second set of relation tags and so forth. The second parameter is \texttt{root}. \texttt{root} states the local minimum root sub-tree that allows the buildGrammar.txtl to locate the specific non-terminal. The third parameter is \texttt{role}. It provides information to the length mutator for mutation. If the length mutator changes the
value of the Content-Length, it may cause the web server to behave anomalously.

Relation tags can also apply to multipart packets; for example, uploading a png file to the server.

Figure 4-9: A partial multipart packet with markup tag “length”

Figure 4-9 shows the partial multipart packet with the markup tag length. Note that this marked up packet contains several null tags in the message body. This is caused by pre-processing of the multipart packet. Without pre-processing of the multipart packet, TXL will fail to completely parse (TXL0191E) this multipart packet when it comes across the first null character. These null tags are markers that replace all the null characters appearing in the message body. After inserting the markup tags, a post-processing of the multipart packet must be done to remove all the null markers and reinsert the original null characters.

4.5 Protocol Description Grammars

There are two kinds of protocol description grammars. The first one is low and/or middle level protocol description grammar. The second one is high level protocol
description grammar. These two description grammar files allow testers to specify which non-terminals of a given grammar should be marked up so as to generate a grammar markup protocol file named httpRequest.lproto. A TXL program called buildGrammar.txl parses this httpRequest.lproto to generate a new TXL program called optAndTag.grm. The optAndTag.grm will then be used by another TXL program called buildMarkup.txl to generate the final InsertMarkup.txl program. This program inserts all the desired tags on the HTTP request messages. Detail discussions will be in Chapter 5 and 6.

4.6 Tag dictionary

The Tag dictionary is a text file which contains all the available markup tags. The purpose of this tags dictionary is for a program named buildGrammar.txl to preprocess all the markup tags which belong to SST. Therefore, the buildMarkup.txl can differentiate between the SST XML tags and the protocol description XML tags. In some situations, we may want to markup the protocol description XML tags for mutation. Figure 4-10 shows the syntaxSpecific tag marks the XML tag from the SOAP protocol. Then we can see a clear distinction between the SST tags with the XML tags from the XML protocol message.

After changing the markup tag format, the buildMarkup.txl knows the <test:responseOK> is the content of the syntaxSpecific markup tag which lies in between the start tag <syntaxSpecific> and the end tag </syntaxSpecific>. Consequently, the misinterpretation of the <test:responseOK> as another markup tag to markup the </syntaxSpecific> to cause program execution failure is avoided.
4.7 Test case generator

The Test case generator is a script to execute mutators to create test cases. A total of six mutators for low/middle level protocol specification and five mutators for high level protocol specification in a first-in-first-out queue wait for an execution. In order to keep track of how many mutants have been created and to determine from which request packet a mutant originated from, a special naming of each mutant is required. Each name of a mutant consists of two parts. The first part is the name of the original packet. The second part is a unique mutant number. The unique mutant number is obtained from a mutant counter. The mutant counter increases by one for each mutant creation. If there are a total of 3 mutants generated by two mutators from a particular packet named Req00002.dat, and a total of 5 mutants generated by three mutators from another packet
named Req00003.dat, then there will be a total of eight mutants and their names shows in Table 4-5.

<table>
<thead>
<tr>
<th>Test cases</th>
<th>Original packets names</th>
<th>Mutated packets names</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Req00002.dat</td>
<td>Req00002.dat.markup-Mutant-1</td>
</tr>
<tr>
<td>2</td>
<td>Req00002.dat</td>
<td>Req00002.dat.markup-Mutant-2</td>
</tr>
<tr>
<td>3</td>
<td>Req00002.dat</td>
<td>Req00002.dat.markup-Mutant-3</td>
</tr>
<tr>
<td>4</td>
<td>Req00003.dat</td>
<td>Req00003.dat.markup-Mutant-4</td>
</tr>
<tr>
<td>5</td>
<td>Req00003.dat</td>
<td>Req00003.dat.markup-Mutant-5</td>
</tr>
<tr>
<td>6</td>
<td>Req00003.dat</td>
<td>Req00003.dat.markup-Mutant-6</td>
</tr>
<tr>
<td>7</td>
<td>Req00003.dat</td>
<td>Req00003.dat.markup-Mutant-7</td>
</tr>
<tr>
<td>8</td>
<td>Req00003.dat</td>
<td>Req00003.dat.markup-Mutant-8</td>
</tr>
</tbody>
</table>

Table 4-5: Naming test cases

Having a good naming convention, it will be very easy to trace the origin of the mutant packet and how it breaks the system. If the server crashes or responds differently after sending the Test case 6 to the server, based on the prefix of this test case, Req00003.dat, then we can look up the original Req00003.dat and the Req00003.dat.markup to find the reason that caused the server to crash.

4.8 Mutation engine

SST consists of a total of six mutators for low and/middle level of protocol specification. Each mutator is designed for a specific markup tag. A mutator is a mutation engine that creates mutants from the original request messages. Each mutant is a variant of the original message with a slight difference. If the mutant is exactly the same as the original, this is called an equivalent mutant. An equivalent mutant is useful in some situations. For example, it can validate whether or not a testing framework is working properly by sending an equivalent mutant to a testing server for the first run. The following discussions show how mutants are created by each mutation engine.
CaseSensitiveTagMutationEngine changes the content of the caseSensitive tag from lower case to uppercase or vice versa. The purpose of this mutation is to break the syntax of any grammar which is under case sensitive limitation. The HTTP request method POST must be in upper case letters according to RFC2616 [19]. Table 4-6 shows how this mutation engine creates mutants.

<table>
<thead>
<tr>
<th>Packet name</th>
<th>Original message</th>
<th>POST /return.asp HTTP/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req00001.dat</td>
<td>&lt;caseSensitive&gt;POST&lt;/caseSensitive&gt; /return.asp HTTP/1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
<tr>
<td>Req00001.dat.markup</td>
<td>POST /return.asp HTTP/1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
<tr>
<td>Req00001.dat.markup- Mutant-1</td>
<td>POST /return.asp HTTP/1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
<tr>
<td>Req00001.dat.markup- Mutant-2</td>
<td>post /return.asp HTTP/1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-6: Two mutants created by CaseSensitiveTagMutationEngine

Although the first mutant is as same as the original message, this is not an implementation error. Consider a string that contains characters with a mix of upper case and lower case, changing all characters to upper case and changing all characters to lower case will break the syntax of case sensitive in any situation.

CharSpecificTagMutationEngine changes a character which is marked by the charSpecific tag to another character. The purpose of this mutation is to break the syntax of any given grammar. Table 4-7 shows how this mutation engine creates mutants.

<table>
<thead>
<tr>
<th>Packet name</th>
<th>Original message</th>
<th>POST /return.asp HTTP/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req00001.dat</td>
<td>POST /return.asp HTTP&lt;charSpecific&gt;/&lt;/charSpecific&gt;1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
<tr>
<td>Req00001.dat.markup</td>
<td>POST /return.asp HTTP/1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
<tr>
<td>Req00001.dat.markup- Mutant-3</td>
<td>POST /return.asp HTTP/1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
<tr>
<td>Req00001.dat.markup- Mutant-4</td>
<td>POST /return.asp HTTP/1.1 Host: 192.168.1.105</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-7: Two mutants created by CharSpecificTagMutationEngine

51
CharSpecificTagMutationEngine can generate any number of mutants. It completely depends on how many rows of character(s) are stored in a special file called Characters. The mutation engine fetches one row of character(s) at a time to replace the original character which is bounded by the charSpecific tag. Figure 4-11 shows the relationship between the mutation engine and the file Characters.

DateSpecificTagMutationEngine changes the date format which is marked by the dateSpecific tag to another date format. The way the dateSpecific mutation engine generates mutants is the same as the charSpecific mutation engine. The number of mutants generated depends on a special file named Dates. Table 4-8 shows how this mutation engine creates two mutants.

<table>
<thead>
<tr>
<th>Packet name</th>
<th>Original message</th>
<th>If-Modified-Since:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req00002.dat</td>
<td>If-Modified-Since: Wed, 18 Feb 2009 7:43:23 GMT</td>
<td></td>
</tr>
<tr>
<td>Req00002.dat.markup</td>
<td>If-Modified-Since: &lt;dateSpecific&gt;Wed, 18 Feb 2009&lt;/dateSpecific&gt; 7:43:23 GMT</td>
<td></td>
</tr>
<tr>
<td>Req00002.dat.markup-Mutant-5</td>
<td>If-Modified-Since: Wed, 2147483648 Feb 2009 7:43:23 GMT</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-8: Two mutants with their dates mutated
The days of two mutants shown in Table 10 were changed from 18 to 128 and 2147483648. If the web server allocates one byte of memory for a signed integer to store a value of a day, then the value 128 will cause this day variable to overflow. The value 2147483648 will cause a 4-byte signed integer to overflow.

SyntaxSpecificTag and ValueLimitationTag mutation engines generate mutants basically in the same way as the dateSpecific and charSpecific mutation engines. Tester prepares specific files for syntaxSpecific and valueLimitation mutation engines to replace the content of these tags.

EnumeratedLiteralTagMutationEngine has a different way to generate mutants. This mutation engine does not require tester to prepare any different characters, strings or numbers to replace the context of the enumeratedLiteral tag. As the name of this tag suggests, mutants are created by enumerating the productions of the tagged non-terminal. Figure 4-12 shows enumeratedLiteral mutation engine enumerating all the methods from the provided grammar.
LengthTagMutationEngine also does not require tester to prepare any characters or strings to replace the content of the length tag. The mutation engine will generate two kinds of mutants. The first kind of mutant will mutate the role of the first terminal. The second kind of mutant will mutate the role of the second terminal. The role is the attribute of the tag. Referring to Figure 4-6, the role of the first terminal is “length” and the content of this length tag is 48. Therefore, the mutation engine will create a total of 6 mutants by a simple algorithm that divides the length in half until a length of ‘0’ is obtained. This is represented in Figure 4-13a. The role of the second terminal is “value”. The mutation engine will create a total of 6 new strings shown in Figure 4-13b. Notice that the last string is depicted with (NULL) which means there is an empty message body although the Content-Length says that the message body should have 48 bytes.

<table>
<thead>
<tr>
<th>Content-Length: 24</th>
<th>Content-Length: 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-Length: 6</td>
<td>Content-Length: 3</td>
</tr>
<tr>
<td>Content-Length: 1</td>
<td>Content-Length: 0</td>
</tr>
</tbody>
</table>

Figure 4-13a: the first role mutants

<table>
<thead>
<tr>
<th>FirstName=John&amp;LastName=</th>
<th>FirstName=Jo</th>
</tr>
</thead>
<tbody>
<tr>
<td>FirstName=JoN</td>
<td>FirstN</td>
</tr>
<tr>
<td>FirstN</td>
<td>F</td>
</tr>
<tr>
<td>(NULL)</td>
<td>(NULL)</td>
</tr>
</tbody>
</table>

Figure 4-13b: the second role mutants

Figure 4-13: The first and the second roles of mutants

4.9 Injector

Injector acts as an organizer to prepare all the necessary steps for sending out request packets and receiving response packets. Detail discussion will be in Chapter 7.
4.10 Travel agents

The Travel agent is responsible for delivering messages to the targeted server and receiving messages back from the server. This is a socket level submodule. It establishes a connection between a client and a server for communication. SST consists of two travel agents. The first one is a generic travel agent. The second one is an HTTP travel agent. If the sent packet is not a mutated packet, the HTTP travel agent will send the packet to the server. If the sent packet is a mutated packet or follows a mutated packet, then the generic travel agent sends the packet to the server with the socket timeout set. In case the server responds abnormally, a socket timeout can prevent the travel agent from waiting indefinitely for the server response. Travel agents can also provide a secure connection, for example, using HTTPS via a secure socket layer to send encrypted messages to the server.

4.11 Oracle

The Oracle is responsible for checking the correctness of the response messages from the travel agent. If there are any discrepancies between a pair of RMB/RMA response messages, then it may suggest the sent mutant was killed. Creating an oracle for this testing method is possible but not a trivial task. This is because we are not only concerned with the server crashing, but also with altering and/or theft of information. This becomes more complex as we have to decide on the acceptable response messages from the server for each different mutated message. Please refer the Chapter 9 (Future Work) for more detail.
The current oracle consists of two phases. The first phase is to check whether the injector has completed each test run. This means all the packets in a test run have been sent to the server. In some situations, the injector will stop the test run after the mutated packet has been sent. This may be because the server is unable to respond to any more requests after receiving the mutated packet. If the test run passes the preliminary check, then the oracle will start a detailed analysis.

A detailed analysis is the second phase and consists of two stages. The first stage compares each character of the original response message to the response message received from the mutated request message. If they are identical, it means the response message received from the mutated request message is well-formed. However, if they are not identical, the oracle cannot make the verdict that the response message received from the mutated request message is well-formed. The oracle will generate the report and the tester needs to analyze this report to make the final decision.

4.12 Summary

This chapter’s discussion focuses mainly on the SST modules. However, the low and middle protocol grammar, and high level protocol grammar were just mentioned in brief in this chapter. Please refer to Chapter 5 and 6 for details regarding these two protocols’ grammars. In addition, more information about the Oracle will be discussed in the Future Work section.
Chapter 5

Syntax-based Security Testing low and middle level concrete architectures

In this chapter, the complete process from preparing test cases to conducting testing will be demonstrated. Figure 5-1 shows the steps for generating test cases. The low/mid level protocol specification file and the description language file will be discussed in detail. This chapter can be used as a user manual to generate test cases from a low level and/or a middle level protocol specification. Generating test cases from a middle protocol specification is the same as from a low level. This chapter will illustrate how to utilize a low level protocol specification as an example to generate test cases.

Figure 5-1: The steps for generating test cases

In Figure 5-1, a tester will create a protocol specification and insert markup tags at the designated non-terminals. From these, SST will generate a combined protocol description language file. Then a build working grammar program parses the combined
protocol description language file to generate the working grammar. This grammar is then used to generate the Insert Markup Program. This generated program marks up the packets and the mutation engines will create mutants according to the markup tags of the packets. The following sections discuss each step in detail.

5.1 Creating the low and middle levels combined protocol description language file

Figure 5-1 shows the steps to generate test cases. After test case generation, the SST starts to run the testing. The whole testing process is automatic except the first step. The first step requests the tester to prepare the generalized HTTP request grammar (protocol specification) and a description language file. A generalized HTTP request grammar is created according to the given text-based communication protocol syntax. This grammar does not represent the full detail information of the specification.

A description file contains the specialized HTTP request grammar and also allows insertion of markup tags surrounding non-terminals. The specialized HTTP request grammar further defines the generalized HTTP grammar (redefined protocol specification). For example, the non-terminal method is either OPTIONS or GET or POST etc according to its specification. Figure 5-2a and 5-2b show the generalized grammar and specialized grammar for the non-terminal method. The method define statement in Figure 5-2a simply uses [id] to represent all the methods. The [id] type in TXL is any string beginning with a letter or underscore and continuing with any combination of letters, digits and underscore.
There are two ways to generate a combined protocol description language file. The first method is to create the combined protocol description language file by hand. The tester modifies the generalized HTTP grammar and adds markup tags to produce the combined protocol description language file. The second method is to use a description file to generate the combined protocol description language file. Figure 5-3 illustrates how to create a protocol description language file. SST provides a merging program that replaces non-terminal definitions in the generalized grammar file with the new versions (including markup) from the description file. Any additional grammar definitions in the description file are also added to produce a combined protocol description language file. The purpose of this second method is to increase the efficiency of the testing framework.
The contents of a description file is very similar to the generalized grammar, the differences lie with a description file containing markup tags and more importantly, the new redefined non-terminals. New redefined non-terminals can make the entire testing process robust. The following three subsections discuss the goal, the specification of description file, and the procedures to generate the combined protocol description language file.

5.1.1 The goal of low and middle levels description file

The low/mid level description file will be used in Syntax-based testing to generate a combined protocol description language file. The SST system can become more efficient with the low/middle level description file in many ways. First of all, without this description file, testers need to understand the entire protocol specification to implement the whole grammar that able to allow testers to insert markup tags. Consequently, it causes more time to develop this grammar file. Second, some of the grammar may not interest the testers. Therefore, there is no point to create this grammar file to fulfill hundred percent of the protocol specifications. Third, even though this grammar file is implemented exactly according to the specifications, this rigid grammar file may not able testers to insert a particular tag at a specific location. Fourth, using description file is more cost-efficient because testers run the testing independently. Each tester can override an individual non-terminal to create new test cases for testing. Without this approach, all testers will use the same copy of the grammar file to generate test cases and all testers will have the same test cases. And last but not least, it is impossible to implement a protocol specification at the fine granular level that is suitable for all testers to drop the
markup tags wherever they want to. In addition, using a description file with TXL supported agile parsing technique [16] is the perfect design for SST to generate test cases.

5.1.2 The specification of low and middle levels description file

The low level description file is a partial grammar of the generalized grammar. Therefore, the specification of the low level description file is the same as the specification of the generalized grammar. This means that the low level description file for the HTTP generalized grammar will not be the same as the low level description file for the FTP generalized grammar. Testers copy parts of the grammar from the generalized grammar file, redefine the definition of some of the non-terminals according to insert the markup tags. The markup tags annotate additional semantics to the non-terminals for mutation. Figure 5-4 depicts the relationships among the generalized grammar, the description file, and the protocol specification.

![Figure 5-4: The relationships among the generalized grammar, the description file, and the protocol specification](image-url)
In Figure 5-4, according to the specification of the HTTP, the version number should begin with HTTP, then follow with a forward slash and the decimal number to one decimal place. Say a tester only wants to break the syntax of the http-version number. First, the tester copies the definition of the non-terminal http-version to create an initial description file. Then the tester redefines this definition of the non-terminal http-version. Redefining this non-terminal is based on the kind of constraints the tester wants to break. After redefining the http-version, a markup tag is inserted to the new non-terminal to annotate additional semantics to the grammar. The creation of the low/mid level description file is now complete and generation of the combined protocol description language file will follow. Figure 5-5 shows the final combined protocol description language file. Note that everything will remain the same except the redefined non-terminal http-version. Please refer Sections 5.4 and 5.5 for detailed discussion on breaking the syntax of the http-version.

...define request-line
    [method][space][request-uri][space][http-version][CRLF]
end define
define http-version
    ‘HTTP ’/[number]<charSpecific>[character]</charSpecific>[number]
end define
define character
    ‘.’
end define
....

Figure 5-5: Final protocol description language file
5.2 Generating working grammar and rules

In this step, the combined protocol description language file is used to generate an Insert Markup Program that will insert the markup from the grammar around all instances in the captured packets. Figure 5-6 shows the steps to create working grammar and rules, and an Insert Markup Program.

![Diagram of generating a new grammar and rules, and an Insert Markup Program](image)

Figure 5-6: Generating a new grammar and rules, and an Insert Markup Program

In Figure 5-6, a build working grammar program parses a combined protocol description language file to generate a working grammar and rules. Then the working grammar and rules will be used to create an Insert Markup Program.

As the example in Figure 5-7b shows the enumerated literal tag and the date specific tag surrounding the non-terminal **method** and the non-terminal **date_string**, respectively. Figure 5-7a shows a partial original HTTP request grammar and Figure 5-7b shows the partial resulting combined protocol description language.
define request_line
  [method] [space] [request_uri] [space] [http_version] [CRLF]
end define

define HTTP_date
  [date_string] [space] [time] [space] GMT
end define

...
The build working grammar program searches the entire combined protocol description language file and replace each of the markup tags with a new non-terminal markable_methodX where X is a unique number. For example the first new non-terminal will be markable_method1. Figure 5-8 shows the working grammar corresponding to the Figure 5-7b. In Figure 5-8, the non-terminal method at the define request_line is replaced by markable_method1 and the non-terminal date_string is replaced by markable_method2. Since the new non-terminals are added, the corresponding definitions of these two new non-terminals must be added (shown in Figure 5-8 after the second ellipsis). This grammar is used by mutators requiring markup in the captured messages. In order to parse the original unmarked captured messages, we need to make the makeup non-terminals optional. Figure 5-9 shows the two new non-terminals markable_method with its redefined definitions.

```
redefine markable_method1
  [method]
  | ...
end redefine

redefine markable_method2
  [date_string]
  | ...
end redefine
```

Figure 5-9: Redefine the markable_method

Referring to Figure 5-8, the request_line is changed from [method] [space] [request_uri]... to [markable_method1] [space] [request_uri]. In Figure 5-9, the redefine statements allow the original unmarked method and date string to be parsed. The bar | at each of the redefine statements mean the beginning of the new alternative pattern and the ellipsis preserves the original pattern. Figure 5-10 presents the
original HTTP_date parse tree and Figure 5-11 depicts the HTTP_date alternative parse tree with the new pattern.

![Original HTTP_date parse tree](image)

**Figure 5-10: The original HTTP_date parse tree**

![Alternative HTTP_date parse tree](image)

**Figure 5-11: An alternative parse tree of HTTP_date**

Once the markup grammar is created, a set of rules is defined to insert the markup into message. This example will have two rules corresponding to the markable_method1 and the markable_method2. Figure 5-12 shows these two rules.

```plaintext
rule insert_markable_method1
   replace $ [markable_method1]
       M1 [method]
       by
       < enumeratedLiteral > M1 < / enumeratedLiteral >
   end rule

rule insert_markable_method2
   replace $ [markable_method2]
       M2 [date_string]
       by
       < dateSpecific > M2 < / dateSpecific >
   end rule
```
Figure 5-12: Two rules for markable_method1 and markable_method2

In Figure 5-12, these two rules will be used by the build markup program to generate the Insert Markup Program. The next section shows that the build markup program generates the Insert Markup Program that contains the rules for markable_method1 and markable_method2.

5.3 Generating the Insert Markup Program

This is the last step to generate the Insert Markup Program. The build markup program parses the working grammar to produce the Insert Markup Program. Figure 5-13 illustrates the complete code of the Insert Markup Program. This program parses a packet and searches the packet parse tree to locate the terminal of markable_method1 and the terminal of markable_method2 in accordance with the two rules. Once the terminal has been found, then this terminal will be replaced by a new terminal.
Figure 5-13: The complete code of the Insert Markup Program

An example of the rule insert_markable_method2; the terminal of the markable_method2 is found from the packet parse tree and its value is stored to the variable M2. Then the M2 will be replaced by <dateSpecific> M2 </dateSpecific>. As a result, the date found in the packet will be marked up. Figure 5-14 shows the date is marked up by the rule insert_markable_method2.

```
include "tagHttpRequest.grm"
include "optTagHttpRequest.grm"
function main
  replace [program]
    p [ program ]
  export gbluid[number]
  1
  by
    p [insert_markable_method1]
      [insert_markable_method2]
  end function

rule insert_markable_method1
  replace $ [markable_method1]
    M1 [method]
  by
    < enumeratedLiteral > M1 </ enumeratedLiteral >
end rule

rule insert_markable_method2
  replace $ [markable_method2]
    M2 [date_string]
  by
    < dateSpecific > M2 </ dateSpecific >
end rule
```

Figure 5-14: The insert_markable_method2 marked up the date
In Figure 5-14, the value of $M_2$ is 03/10/1900. The replacement is <dateSpecific> 03/10/1900</dateSpecific>. It is worth mentioning that the entire process of test cases generation is protocol independent.

The Insert Markup Program marks up each packet if and only if the packet contains the terminal(s) corresponding to the non-terminal(s) being marked up from the the combined protocol description language file. Packets being marked up by the Insert Markup Program are the instances of the combined protocol description language file. Figure 5-15 shows the relationship among a description file, a combined protocol description language file, the HTTP captured request packets, and the HTTP marked up packets (the instances of the meta-markupTag). In Figure 5-15, the build markup program generates the Insert Markup program. Then the Insert Markup program parses the HTTP request packets, for example, Req00001.dat as an input source for transformation. The Insert Markup Program looks for whether or not this packet contains any terminals specified in the combined protocol description language file. If yes, then Insert Markup Program inserts the corresponding markup tags surrounding those terminals. Section 5.4 demonstrates how to generate test cases by using this Insert Markup Program and how the markup tags are related to the description file and the grammar.
Figure 5-15: The relationship among a description file, a combined protocol description language file, the HTTP captured request packets, and the HTTP marked up packets.
5.4 Relationship between test cases and the description file

To generate test cases from the grammar, testers must create an httpRequest.lproto file first. Testers can manually insert the markup tags at the generalized HTTP grammar to build the httpRequest.lproto file. For example, if testers want the InsertMarkup.txl program to insert an enumeratedLiteral markup tag at any HTTP request packets that contain the terminal of method e.g. `<enumeratedLiteral>GET</enumeratedLiteral>`, then testers simply drop an enumeratedLiteral tag at the non-terminal of method. Figure 5-16 illustrates the enumeratedLiteral tag manually inserted at the non-terminal of method.

```plaintext
define program
  [request-message]
end define

define request-message
  [request-line][repeat headers_message][CRLF][opt message_body]
end define

define request-line
  <enumeratedLiteral>[method]</enumeratedLiteral>[space][request-uri][space][http-version][CRLF]
end define

define method
  OPTIONS | GET | HEAD | POST | PUT | DELETE | TRACE | CONNECT
end define

define headers_message
  [general_header][CRLF]
  | [request_header][CRLF]
  | [entity_header][CRLF]
end define
```

Figure 5-16: Manually inserted enumeratedLiteral tag at the non-terminal of method

Before the enumeratedLiteral tag was added, this file was a generalized HTTP grammar. As noted in section 5.1, the combined protocol description file can also be generated. In addition to providing markup, this file can also provide a different parse for
parts of the message. The different parse may provide more structure to those parts of the message than the original grammar, giving the test greater flexibility in placing the markup. By combining this description file with the TXL supported agile parsing, the generalized grammar no longer needs to be precise to represent the grammar of each particular text-based protocol under testing. As a result, building the initial generalized grammar model to test another text-based protocol requires less time. Testers can then focus their time on the testing area of interest. The following example demonstrates how to prepare a description file.

For example testers may want to mutate the HTTP version number from 1.1 to 2.1 (or any arbitrary number). Referring to Figure 5-17, the non-terminal of http-version appears at the definition of request-line, and the definition of the non-terminal of http-version is HTTP / [number].

```plaintext
define request-line
   [method][space][request-uri][space][http-version][CRLF]
end define

define http-version
   HTTP / [number]
end define
```

Figure 5-17: The definitions of the non-terminal of http-version

Because the goal is to mutate the version number, a valueLimitation tag must be inserted onto the non-terminal of number at the definition of the non-terminal http-version. Therefore, the description file must contain a new definition for http-version. Figure 5-18 shows a new definition of the non-terminal of http-version.
Figure 5-18: A new definition of the non-terminal of http-version

Assuming Figure 5-19 is a HTTP request packet named Req00001.dat, Insert Markup Program parses this request packet to insert the valueLimitation tag onto the packet. Figure 5-20 represents the request packet Req00001.dat after inserted the valueLimitation tag.

```
POST /received.html HTTP/1.1
Host: 192.168.0.2:8080
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220
BonEcho/2.0.0.11
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8
image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.0.2:8080/
Content-Type: application/x-www-form-urlencoded
Content-Length: 19
\r\nfname=Ben&lname=Kam
```

Figure 5-19: Req00001.dat - a HTTP request packet before marked up

Now, assume testers want to mutate the decimal point of the http-version number to be another character, for example, change \texttt{HTTP/1.1} to \texttt{HTTP/1:1}. Testers simply change the definition of the non-terminal of http-version in the description file. Figure 5-21 shows a new, more thorough definition of the non-terminal http-version. Figure 5-22 shows the charSpecific tag added to the newly defined http-version to create a new description file.
Figure 5-20: Req00001.dat.markup – a HTTP request packet after marked up with valueLimitation tag

```plaintext
POST /received.html HTTP/<valueLimitation>1.1</valueLimitation>
Host: 192.168.0.2:8080
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220
BonEcho/2.0.0.11
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8
image/png,/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.0.2:8080/
Content-Type: application/x-www-form-urlencoded
Content-Length: 19

fname=Ben&lname=Kam
```

Figure 5-21: A new definition of the non-terminal http-version

```plaintext
define http-version
    HTTP / [number] [period] [number]
end define

define period
    '.
end define
```

Figure 5-22: A new description file is created

```plaintext
define http-version
    HTTP / [number] <charSpecific> [period] </charSpecific> [number]
end define

define period
    '.
end define
```

Figure 5-23 represents the request packet Req00001.dat after inserting the charSpecific tag.
Before showing how to obtain test cases by using the HTTP request marked up packets, the following is one more example to show how to add nested markup tags to a HTTP request packet. Assume testers want to mutate the non-terminals of method by changing its semantic and syntax. Figure 5-24 shows how these tasks can be done by adding two tags nested together.

```
define request-line
  <caseSensitive><enumeratedLiteral>[method]</enumeratedLiteral></caseSensitive>
  [space][request-uri][space][http-version][CRLF]
end define
```

In Figure 5-24, there are two tags nested together for the mark up the non-terminal [method]. The ordering of these two tags does not make any difference to test cases generation. The content of the caseSensitive mutator is the terminal of method and so is the enumeratedLiteral mutator. However, the structure of the nested tags must be well-formed and cannot be crossed. Please refer to Chapter 3.4 for details. Figure 5-25 represents the request packet Req00001.dat after the nested tags are added.
5.5 Using grammar to generate test cases

After three examples of creating a description file, we now discuss how to create test cases by using the marked up packets. Creating test cases can be done by grammar or test cases libraries. This subsection shows how to create test cases by grammar.

The enumerated literal tag uses grammar to generate mutants. Using the grammar to generate mutants can be quite complicated. The enumerated literal mutation engine is dynamically generated by a special program. The special program analyzes the marked up grammar. The program then extracts all the alternative terminals from the marked up non-terminal to write a set of transformation rules and create the enumerated literal mutator. For example, assume the HTTP request method is marked up by the enumerated literal tag. The special program will create eight rules in accordance with the marked up grammar. Figure 5-26 shows the relationship between the alternative terminals from the marked up non-terminal and eight rules generation.
In Figure 5-26, each rule has one parameter which is the marked up packet file name. This file name is the base name to name all enumerated literal mutants. For example, if the file name is Req00001.dat.markup, then the name of the first mutant will be Req00001.dat.markup-mutant-1. The enumerated literal engine consists of a general program that executes each rule and may generate a mutant. In Figure 5-26, the Req00001.dat.markup packet contains the enumerated literal tag surrounding the method \texttt{POST} which will be used in Figure 5-27 to show how mutants are generated according to the rules.
In Figure 5-27, the first mutant is Req00001.dat.markup-mutant-1. According to the R1:enumeratedMethodCONNECT, the alternative request method is CONNECT. Therefore, the POST method of Req00001.dat.markup will be replaced by this alternative method thus creating the first mutant. Note that the R5 has no mutant creation because the alternative request method is POST which is the same as the Req00001.dat.markup.

Referring the Figure 5-25, a HTTP request packet Req00001.dat.markup is created by the Insert Markup Program which contains the nested caseSensitive and
enumeratedLiteral tags. Figure 5-28 shows the mechanism to generate different kinds of mutants according to different kinds of mutators.

![Figure 5-28: The mechanism to generate different kinds of mutants](image)

In Figure 5-28, all the HTTP request marked up messages are fetched by a script and sends it to a vector of mutators. Each marked up request message will be checked to see if it contains the corresponding markup tag of the mutator. If the corresponding markup tag is found, then the mutator generates test cases. Otherwise, the request message will pass on to the next mutator to check if the request message contains its tag. The Req00001.dat.mark request packet from Figure 5-25 generates the 21 test cases, as shown in Table 5-1.

Table 5-1 does not show the whole mutated request message. It only shows the mutated line. To be more precise, the mutated line still has the nested tags attached.
Figure 5-29 and Figure 5-30 illustrate the actual mutated packet and the final mutant, respectively.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>enumeratedLiteral mutants</th>
<th>caseSensitive mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CONNECT/received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>2</td>
<td>TRACE /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>3</td>
<td>DELETE /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>4</td>
<td>PUT /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>5</td>
<td>HEAD /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>6</td>
<td>GET /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>7</td>
<td>OPTIONS /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>8</td>
<td>connect /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>9</td>
<td>CONNECT /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>10</td>
<td>trace /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>11</td>
<td>TRACE /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>12</td>
<td>delete /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>13</td>
<td>DELETE /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>14</td>
<td>put /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>15</td>
<td>PUT /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>16</td>
<td>head /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>17</td>
<td>HEAD /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>18</td>
<td>get /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>19</td>
<td>GET/received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>20</td>
<td>options /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
<tr>
<td>21</td>
<td>OPTIONS /received.html HTTP/1.1\n</td>
<td>\n</td>
</tr>
</tbody>
</table>

Table 5-1: A total of 21 test cases generated from a nested tags packet
In Figure 5-29, the mutated packet is not the final test case because the mutated packet still has tags attached. The last step is to remove all the markup tags yielding the final mutants shown in Figure 5-30. The next subsection introduces how to generate test cases by markup tag libraries.
5.6 Using markup tags libraries to generate test cases

The procedures of generating test cases by grammar and by markup tags libraries differ slightly. The mutator is simply an external program that uses a parameter file to accomplish the mutation. Referring to Figure 5-23, the marked up packet Req00001.dat.markup contains the charSpecific tag at the line of POST /received.html HTTP /1<charSpecific>.

If the characters ~ ! @ # $ % ^ & * , ; : are in the charSpecific library, then Table 5-2 shows all the test cases that can be created corresponding to this library.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>charSpecific mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>POST /received.html HTTP/1~1\r\n</td>
</tr>
<tr>
<td>2</td>
<td>POST /received.html HTTP/1!1\r\n</td>
</tr>
<tr>
<td>3</td>
<td>POST /received.html HTTP/1@1\r\n</td>
</tr>
<tr>
<td>4</td>
<td>POST /received.html HTTP/1#1\r\n</td>
</tr>
<tr>
<td>5</td>
<td>POST /received.html HTTP/1$1\r\n</td>
</tr>
<tr>
<td>6</td>
<td>POST /received.html HTTP/1%1\r\n</td>
</tr>
<tr>
<td>7</td>
<td>POST /received.html HTTP/1^1\r\n</td>
</tr>
<tr>
<td>8</td>
<td>POST /received.html HTTP/1&amp;1\r\n</td>
</tr>
<tr>
<td>9</td>
<td>POST /received.html HTTP/1*1\r\n</td>
</tr>
<tr>
<td>10</td>
<td>POST /received.html HTTP/1,1\r\n</td>
</tr>
<tr>
<td>11</td>
<td>POST /received.html HTTP/1;1\r\n</td>
</tr>
<tr>
<td>12</td>
<td>POST /received.html HTTP/1:\r\n</td>
</tr>
</tbody>
</table>

Table 5-2: A total of 12 test cases generated according from the library

Note that Table 5-2 only shows the mutated line of the test cases. The number of test cases generated by using a library can be limitless. The benefit of this approach will be discussed in detail in Chapter 6. It will show the importance and robustness of this method to create mutants.
Chapter 6

Syntax-based Security Testing high level concrete architectures

In this chapter, the high level description file will be discussed in detail, followed by a demonstration of test case generation. This file contains patterns used to identify the type of the message and to insert markup specific to the type of the message. The purpose of the high level description file is to mark up classified messages, for example, user logon packets and shopping cart packets. The structure of the high level description file is completely different than the low/mid level description language file. The high level description language file specifies high level (application level) text-based protocol properties, for example, a shopping cart protocol. The application level specification primarily focuses on the business rules. The markup tags break the constraints of the business rules rather than breaking the text-based protocol specification. This chapter can be used as a user manual to generate test cases from a high level protocol specification.

6.1 The goal of high level description language

The goal of the high level description language file is to give complete instructions to insert markup tags at the application level. First, it tells the SST what kind of grammar is being used as the base grammar, for example, the http request grammar (httpRequest.grm). Second, it gives instructions to override or extend the definitions of the base grammar non-terminals. Third, it specifies where to insert the markup tags. Fourth, it describes patterns in the application message which is used to identify this packet in the request message packets sequence. Last, it states explicitly the constraints of the user input string in accordance with the business rules.
6.2 The specification of high level description language

A high level description language file contains three sections. Each section is divided by an opening keyword and closing keyword. Figure 6-1 illustrates the first section of a high level description language file. In order to provide distinction, the metasyntax from the figures and text of this section will be displayed in bold text and courier font, respectively.

```
ProtocolGrammar
  uses "httpRequest.grm"
end ProtocolGrammar
```

Figure 6-1: The first section of the high level description language file

The first section is very simple. It instructs the SST to use the httpRequest.grm as the base grammar. A high level protocol specification (referring to Chapter 4.3) is an extension of the low and/or middle level protocol specification. The purpose of this section is to make the description file portable. Testers simply insert the base grammar file name inside the double quotes. It can use any text-based protocol whenever the application level is applicable.

In the second section of the description file, the non-terminals will override or extend the definitions of the existing non-terminals. They may also introduce new non-terminals for use in the extensions. An example of a second section is illustrated in Figure 6-2. This figure shows a new non-terminal named new_message_body. It provides an alternate parse pattern for the original definition of message_body. Figure 6-3 represents the partial HTTP request grammar and shows the original definition of message_body.
In Figure 6-2, the metasyntax of beginning and ending of this second section are **AdditionalGrammar** and **end AdditionalGrammar**. All new non-terminals are placed within this area and will provide alternative parse pattern for the existing non-
terminals. In Figure 6-3, the original non-terminal `message_body` is defined as `repeat token_or_key`. The definition of `token_or_key` is either a token or a key will be accepted as a valid token during the parsing phase. It can parse any lines of characters without any problems e.g. `username=benkam&password=abcd%40%24%24&submit=login`. The advantage lies in the fast implementation of the initial generalized grammar. However, the existing definition of `message_body` has no structure; it is nothing more than a sequence of characters. This line contains user name and password in the form of name and value pair. Testers are unable to insert tags around them by using the existing grammar. In order to allow testers to insert tags around the user name and password, the value of the non-terminal `message_body` will be used as an input source to construct a new variable with the `new_message_body` type. A snippet of code in Figure 6-4 shows how it works.

```
replace $ [message_body]
  Body [message_body]
construct NewBody [opt new_message_body]
  [reparse Body] [addTags]
construct Replacement [message_body]
  [reparse NewBody]
by
  Replacement
```

**Figure 6-4: Replace the content of message_body with new definition**

Assuming a variable of `Body` with original `message_body` type in Figure 6-4 contains the value of `username=benkam&password=abcd%40%24%24&submit=login`. It is impossible to extract the user name and password from the `Body` and drop tags surrounding them based on the existing definition of `message_body`. In order to achieve this goal, first, a new variable `NewBody` with `new_message_body` type must be constructed, then the built-in TXL function “reparse” is used to parse the
content of Body to the NewBody. After reparsing is complete, the function, addTags, calls the deconstructor to break down the content of the NewBody to yield the “paramName=ParamValue”; therefore, testers can drop tags around the “ParamValue”. In this example, the value of NewBody ends up containing username=<caseSensitive>benkam</caseSensitive>&password= <caseSensitive>abcd%40%24%24</caseSensitivity>&submit=login. The final step is to construct a new variable named Replacement with message_body type, then reparse the NewBody to the Replacement and replace the variable Body. As a result, the final parse tree still maintains the same grammatical structure but the content is changed with tags. This approach ensures that the tester does not introduce an ambiguity such as missing the end of line in a MIME header.

The third section of the description file is relatively more complicated than the previous two sections. The metasyntax of the beginning and the ending of this section are Packet and end Packet. There are two parts contained inside this section. The first part describes a pattern of a particular packet. The second part conveys the particular packet properties. Properties contain additional grammar instructions and parameters information. Figure 6-5 shows the third section of a description language file. The first line of this section is Packet “Login”. The “Login” is the name of a packet which is given by a tester. It can be any name but should be meaningful. Because this packet is responsible for user login, it is reasonable to call this packet “Login”.
Figure 6-5: The third section of the high level description language file

Packet "Login"
  Pattern
    "request_uri" matches "cmd=login"
  end Pattern

Properties
  AdditionalGrammarInstruction
    AG1: defines "new_message_body" uses to redefine "message_body"
    AG2: drops tags at "ParamValue"
    AG3: "ParamNameValuePair" is a nonterminal of a new parse pattern
  end AdditionalGrammarInstruction

ParameterInfo
  "username" in "message_body" matches "username="
    ParamValueConstraints
      CaseSensitive "yes"
      Length "1-20"
      LegalChars "a\d"
      StringSpecific "*"
    end ParamValueConstraints
  end ParameterInfo

ParameterInfo
  "password" in "message_body" matches "password="
    ParamValueConstraints
      CaseSensitive "yes"
      Length "2-30"
      LegalChars "!@#$%&_\a\d"
      StringSpecific "*"
    end ParamValueConstraints
  end ParameterInfo

end Properties
end Packet

After the first line, the pattern of the login packet will be discussed. For this example, Figure 6-6 shows that the packet contains a line POST /login/?cmd=login HTTP/1.1. According to the HTTP request grammar shown in Figure 6-7, the non-terminal of Request_line = Method SP request-uri SP HTTP-Version CRLF. The request-uri is /login/?cmd=login which is a unique pattern for recognizing that the packet is a user login packet. Note that in some cases, a simple pattern may not be able to identify the packet.
Consequently, three types of patterns identification are implemented. The first one is simply a single pattern like the example above. The other two contain more than one pattern. Each pattern conjuncts with either an “or” operator or an “and” operator to form composite patterns. Having this setup, allows for identification of any patterns of the packets. Figure 6-8 presents the customized packet. The customized packet is responsible for allowing the user to change the title of the page. In order to accurately describe the patterns of this packet, non-terminals method and request-uri should be used. Figure 6-9 shows using the “and” operator to describe the pattern of a customized packet.
In Figure 6-9, there is more than one pattern description regarding the customized packet. Therefore, a tester can use either the “or” operator or the “and” operator to join them together to form a composite pattern. A composite pattern can increase the accuracy of packet identification.

The second part of the third section of the description file describes the properties of the packet. The first property states all the necessary instructions on how to use the new grammar. Referring to Figure 6-5, AGI: defines “new_message_body” uses to redefine “message_body” instructs the TXL program to use the new non-terminal of new_message_body to redefine the existing non-terminal of...
message_body. AG2: drops tags at “ParamValue” further instructs the TXL program to only drop tags at “ParamValue”. AG3: “ParamNameValuePair” is a nonterminal of a new parse pattern finally instructs the TXL program that the non-terminal ParamValue is derived from the non-terminal ParamNameValuePair. A total of three additional grammar instructions should be sufficient to instruct the TXL program on how to use new non-terminals.

The second property is the parameter information. Parameter information describes the characteristics and constraints of users' input data. The common abstraction of higher level protocols modeled in our language is that of parameters and values. Browsers map input widgets to parameter value pairs, other protocols such as SMTP and FTP also use parameter value pairs to transmit values. Web browsers provide many different ways of allowing users to input data, for example, check box, selection box or input box. Without loss of generality, the input box will be used here as an example. Assuming the login page provides two input boxes for users to input their user name and password. Then the description file should contain a total of two parameters of information.

Referring to Figure 6-5, the first parameter information begins with a keyword, ParameterInfo, and ends with two keywords; end ParameterInfo. The first line of the ParameterInfo username in message_body matches username= instructs the TXL program to find the variable name of the user name. To be more precise, username is a variable for the TXL program to store the value of user name. In this example, username has the value of benkam. On the other hand, username= is the variable of the user name for the web application. If the web
application uses uname as the user name variable, then the ParameterInfo would be
username in message_body matches uname=. Of course, the message_body will be
uname=benkam&password=abcd%40%24%24&submit=
Login instead of username=benkam&password=abcd%40%24%24&submit=
Login. After pattern matching, the value of the username is benkam, in accordance
with the grammar of ParamValue, is [repeat anything_but_not_space] and the
anything_ but_not_space is [not ‘&] [not space] [token].
Once the TXL obtains the username value, tags can be dropped around it for mutation.

Following the first line of ParameterInfo is a
ParamValueConstraints. This subsection describes the constraints of the input
string. Table 6-1 shows the constraints of the user name input string.

<table>
<thead>
<tr>
<th>Type of constraints</th>
<th>Limitations</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaseSensitive</td>
<td>“yes”</td>
<td>It is case sensitive</td>
</tr>
<tr>
<td>Length</td>
<td>“1-20”</td>
<td>Min 1 char, Max 20 chars</td>
</tr>
<tr>
<td>LegalChars</td>
<td>“a’d”</td>
<td>‘a - any one of a-z or A-Z, ‘d – any one of 0-9</td>
</tr>
<tr>
<td>StringSpecific</td>
<td>“*”</td>
<td>Asterisk – wild card, can be anything</td>
</tr>
</tbody>
</table>

Table 6-1: The constraints of the user name input string

In Table 6-1, the CaseSensitive constraint can be either “yes” or “no”. The Length
constraint states the minimum length and the maximum length of the input string. The
LegalChars enforces what the legal input characters are. Lastly, the StringSpecific
accepts any input strings. Detail discussions will be in Subsection 5.3.

The second ParameterInfo describes the characteristics of the password and
its limitations. The first line password in message_body matches password=
would serve the same purpose as the first ParameterInfo of username and the
ParamValueConstraints describes the password input string limitations.
6.3 Packet analyzer

Typically, an online purchasing web site allows customers to browse their products, add items to the shopping cart. If customers want to buy their selected items, they are required to login to the system, then follow several steps such as entering credit card information and mailing addresses. The whole bundle of work flow complies with business rules to form a new set of protocol. This thesis will refer to this new set of protocol as a ‘shopping cart protocol’. When customers enter their user identities and passwords on the login page, the user identities and passwords will have their own specifications in accordance with the shopping cart protocol. Therefore, testers can break the user identities and/or passwords constraints to test the system. However, testers must know which packet is responsible for login before they can mutate the packet. After browsing, adding items to the shopping cart and logging in, there will be over a hundred packets in the captured request message sequence. It is impossible for testers to read each packet one by one manually to identify which one is the login packet. An automated packet analyzer is used to assist testers to identify a particular packet.

The packet analyzer can identify individual packet types by using the high level description language file. For example, the captured message sequence contains 200 packets which includes packets for a user login to the system and a mailing address information submission. After using the packet analyzer, the login packet and mailing packet could be found among these 200 packets, for instance; packet 120 and 165 respectively. Then the system automatically inserts appropriate markup tags into the identified packets. The beginning of the third section of the high level description
language file (Pattern) is used for analyzing packet types. The following section discusses this file in detail.

6.4 Using program to generate test cases

In a high level protocol specification, SST provides a total of four mutators for generating test cases which are case sensitive mutator, length mutator, legal characters mutator, and string specific mutator. The procedures of generating test cases in a high level model are more complicated than the low/middle level. First of all, for the example of web application testing, the HTTP low/middle level protocol specification is static, meaning the protocol specification can be reused to test other web applications. In contrast, with the high level protocol specification, the specification is not static, meaning the protocol specification cannot be reused to test other web applications. For example, a high level protocol specification for a hotmail login page cannot be reused to test an ebay login page. The simplest reason for this is that the names of the variables such as user name and password may not be the same. Therefore, the high level description file is the core architecture to make the high level protocol specification able to create test cases automatically. Figure 6-10 shows the initial step of packet identification in order to generate test cases for a high level protocol specification.
In Figure 6-10, a packet recognizer program extracts information from the high level description language file to generate a new program which can find particular packets e.g. a packet responsible for a user login. This program is a packet analyzer. It analyzes each packet one by one in order to find the login packet. The login packet is black in color and resides in the message sequence. Once the login packet is found, the packet number will be added to the database. The packet analyzer continues to scan through the remaining packets in the sequence since it is possible to have more than one login packet. Note that the created program name and its functionality are dependent on the high level description language file. Referring to the first line of Figure 6-5, the Packet “Login” means the packet is the login type packet. Therefore, the packet recognizer program creates the program that can find the Login packet. Referring to the first line of Figure 6-9, the Packet “Customize” means the packet is responsible for a user to customize his/her home page. Therefore, the packet recognizer program will create a program that can find a customize packet in the captured messages sequence.
Figure 6-11 shows the process of application level packet markup. SST provides a special program extracts the first line of the third section from the high level description language file to dynamically generate a markup packet program. This program is able to drop markup tags at a particular packet which is specified at the first line of the third section of the high level description file. If the first line of the third section is Packet “Login”, then the new generated program will be named dropTagAtLoginPacket.txtl that is able to insert markup tags at the Login packet. If it is Packet “Customize”, then the output program will be dropTagAtCustomizePacket.txtl that is able to insert tags at the Customize packet. Figure 6-12 shows the login packet Req00055.dat after marked up.

After the login packet is marked up, mutators can start generating test cases. Test cases will be generated by using library or without library. In Figure 6-12, all mutants related to the markup tags CaseSensitive, Length, and LegalChars will be generated without library. In addition, mutants related to StringSpecific will be created by using the library.
The case sensitive mutation engine checks the attribute of the CaseSensitive tag. If the value of the parameter P is “yes”, then the mutator changes the content of the CaseSensitive tag from lowercase to uppercase or vice versa. Table 6-2 shows the mutants after the mutation of the marked up packet.

<table>
<thead>
<tr>
<th>Packet name</th>
<th>Original message</th>
<th>username=benkam&amp;password=abcd%40%24%24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req00055.dat</td>
<td>username=(&lt;CaseSensitive p=&quot;yes&quot;&gt;benkam&lt;/CaseSensitive&gt;)&amp;password=(&lt;CaseSensitive p=&quot;yes&quot;&gt;abcd%40%24%24&lt;/CaseSensitive&gt;)&amp;submit=Login</td>
<td></td>
</tr>
<tr>
<td>Req00055.dat.markup</td>
<td>username=(&lt;CaseSensitive p=&quot;yes&quot;&gt;benkam&lt;/CaseSensitive&gt;)&amp;password=(&lt;CaseSensitive p=&quot;yes&quot;&gt;abcd%40%24%24&lt;/CaseSensitive&gt;)&amp;submit=Login</td>
<td></td>
</tr>
<tr>
<td>Req00055.dat.markup-Mutant-1</td>
<td>Username=BENKAM&amp;password=ABCD%40%24%24</td>
<td></td>
</tr>
<tr>
<td>Req00002.dat.markup-Mutant-2</td>
<td>Username=benkam&amp;password=abcd%40%24%24</td>
<td></td>
</tr>
</tbody>
</table>

Table 6-2: Mutants from the case sensitive mutation engine

The length mutation engine reads the attribute of the Length tag to obtain the minimum and maximum length values. Two mutants are created by a simple algorithm. The first mutant is a new string with one character shorter than the minimum requirement. The second one is a new string created by concatenating the input string
itself until the length exceeds the maximum value. Table 6-3 shows two mutants created by the length mutation engine.

<table>
<thead>
<tr>
<th>Packet name</th>
<th>Original message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req00055.dat</td>
<td>username=&lt;Length p=&quot;1-20&quot;&gt;benkam&lt;/Length&gt;</td>
</tr>
<tr>
<td></td>
<td>&amp;password=&lt;Length p=&quot;2-30&quot;&gt;abcd%40%24%24&lt;/Length&gt;</td>
</tr>
<tr>
<td>Req00055.dat.markup</td>
<td>username=&quot;benkam&quot;&amp;password=&quot;abcd%40%24%24&quot;</td>
</tr>
<tr>
<td>Req00055.dat.markup-Mutant-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Username=&quot;a&quot;</td>
</tr>
<tr>
<td>Req00002.dat.markup-Mutant-2</td>
<td>username=&quot;benkambenkambenkambenkam&quot;&amp;password=&quot;abcd%40%24%24&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-3: Two mutants created by the length mutation engine

The legal characters mutation engine requires three external files to generate test cases. The first file named allLegalChars.txt is a library that stores all the printable characters and is created manually (let P be the set of allLegalChars.txt). The second file named LegalChars.txt contains all the legal characters according to the attribute p of the tag <LegalChars p="^\a\d"> (let Q be the set of LegalChars.txt). The third file named IllegalChars.txt contains all the illegal characters and is created by a java program called illegalChars.class (let R be the set of IllegalChars.txt). All the illegal characters generated by illegalChars.class are all the elements belonging to R, where R = P\Q. Each illegal character will replace original input string characters at three different positions (beginning, in between and end), one at a time, to generate three mutants. The positions are the first character (beginning), the second character (between), and the last character (end) of the input string. Table 6-4 shows a total of six mutants generated by the legal characters mutation engine with two illegal characters. The next section discusses mutants generated by using the library.
6.5 Using markup tags libraries to generate test cases

For the high level protocol specification, there is only one mutation engine, string specific mutation engine, which uses the library to create mutants. This mutation engine plays a crucial role in the SST. It creates mutants by replacing the original input string with a specially crafted string or any sequence of random characters. Therefore, the library for this mutation engine collects enriched specially crafted strings from prior literature. In other words, once SST has completed security testing of the web applications, it can be guaranteed that the applications will not be broken by any of the known tricks. In the case that a new form of an input string is able to break the system and is reported, the SST will easily reveal this new form of attack by simply adding the new form of the input string to the library. Table 6-5 shows a couple of mutants with a specially crafted string.
In Table 6-5, the two mutants are created by the strings specific mutation engine using the library. The first one tests whether or not the application can be attacked by SQL injection. The second one tests whether the application can be attacked by command injection. However, the current version of SST requires the tester to locate the string “; finger” from the library and add it to the end of the original input string. This is necessary because the command injection always piggybacks the user input (the original input string) and injects some system commands to the targeted system to steal information.

6.6 Chapter Conclusion

This chapter discusses the details of concrete architecture of a high level protocol specification. It shows the importance of the high level description language file. Although using the description file requires a bit of a learning curve, the time invested with another approach to generate test cases would be outrageous. A template of this high level description file is shown in Figure 6-13. In summary, testers only need to fill out the relevant information inside the double quotes in the first and the third sections, and the new grammar in the second section.
ProtocolGrammar
uses ""
end ProtocolGrammar

AdditionalGrammar
% enter all new non-terminals definition here
end AdditionalGrammar

Packet ""
  Pattern
  "" matches ""
  % optional or (can more than one “or” clause)
  % delete it if not applicable
  or
  "" matches ""
  % optional and (can more than one “and” clause)
  % delete it if not applicable
  and
  "" matches ""
end Pattern

Properties
  AdditionalGrammarInstruction
  AG1: defines "" uses to redefine ""
  AG2: drops tags at ""
  AG3: "" is a nonterminal of a new parse pattern
end AdditionalGrammarInstruction
% One ParameterInfo for one input string, can be more than one ParameterInfo

ParameterInfo
  "" in "" matches ""
  ParamValueConstraints
    CaseSensitive ""
    Length ""
    LegalChars ""
    StringSpecific "#\n"
end ParamValueConstraints
end ParameterInfo
end Properties
end Packet

Figure 6-13: Template for a high level description file
Chapter 7

Injector

The Injector is responsible for replaying message sequences. The injector does not retain any information about the syntax and semantics of the sent message. In the case of a request, it simply fetches a test case (a mutated request packet) from a test suite and sends the test case to the travel agent for delivery. However, if the test case is required to maintain its system state by one or more cookies, the injector will refresh the cookies before sending the test case to the travel agent. For example, some pages or applications only allow users access after login to the system. In order to replay the request messages to access these pages or applications, the system state must be maintained in real time. The injector will utilize the RealTimeUpdate instruction file in order to maintain the state of the HTTP request in real time. The RealTimeUpdate file contains the records of which response messages set cookie(s) to which request messages. Therefore, when an injector replays the request messages, the injector looks up this instruction file to determine whether or not refreshing cookie(s) is required for this message. If refreshing instruction(s) is in place, then the injector will refresh the cookie(s) before handing over the request to the travel agent. Figure 7-1 shows the processes of refreshing cookie(s).
In Figure 7-1, the injector fetches the 5th test case and performs a real time update check before passing it to the travel agent. The real time update process looks up the real time update instruction file and finds an instruction that states this test case has a cookie set by the 2nd response packet. Therefore, the injector uses the regular express function to copy the fresh cookie from the real time response message of the 2nd response packet and replaces the stale cookie in the 5th test case. Real time update instruction files are created by cookie dependency programs. These programs analyze all the request and response packets, then generate a real time update instruction file for the injector in real time that refreshes cookies to maintain the system states if necessary.

7.1 A RealTimeUpdate instruction file

A RealTimeUpdate instruction file contains all the state dependencies between the request message sequence and the response message sequence. This instruction file is created by a TXL program called cookie dependency program with two TXL programs.
get HTTP cookie program and set cookie list program for preprocessing. Figure 7-2 shows the creation of a RealTimeUpdate instruction file.

![Diagram of the creation of a RealTimeUpdate instruction file]

In Figure 7-2, the get HTTP cookie program searches the request messages one by one to find if there are any cookies contained in the packets and if there are, lists them on the cookie list. Similarly, the set cookie list program searches all the response messages to find any set cookies contained in the packets and lists them on the set cookie list. After the cookie list and set cookie list are prepared, the cookie dependency program matches cookies patterns between the cookie list and the set cookie list files in order to find cookie dependencies. Request message cookie dependency may involve more than one response message. For example, the cookie dependency of Req00007.dat request message depends on Resp00002.dat-headers.txt and Resp00006.dat-headers.txt files to set cookies. The RealTimeUpdate instruction file may not be able to handle all situations of maintaining the system state. Please refer the Future Work section for details.
Injector also maintains a database server state. If required, the injector will initialize the state of the database before sending any requests to the server. The last component of the injector is a re-write engine. The injector simply accepts the response messages from the travel agent and stores them in the response directory named Response-Message-After (RMA). If the messages are compressed, then the injector will decompress the messages before storing them to the RMA. If the response messages are chunked, then the injector will merge the messages before storing them to the RMA.
Chapter 8

Experiments in Syntax-based Security Testing

After discussing the SST framework in detail, we now demonstrate the performance and effectiveness of SST by conducting four test cases on two separate protocols. In Section 8.1, the first test case shows the correctness of SST. In Section 8.2, the second test case shows the successful discovery by SST of a vulnerability of a commercial web application. In Section 8.3, the third test case demonstrates protocol independency by showing a new vulnerability in the open source application kOrganizer detected by SST. The last test case illustrates the flexibility of SST by mimicking the PROTOS testing (co-http-reply). The organization of this chapter first states the scope of the first test case and the nine testing results on a couple of toy web servers. The demonstration of the other three cases will then be discussed, followed by the conclusion of this chapter.

8.1 Baseline

A couple of toy servers were constructed that contained vulnerabilities for all nine tags. These servers were used to validate the functionality of the framework before attempting to find new, unknown errors in other applications. A total of nine small tests were conducted to demonstrate nine different mutated packets that were sent to the toy server successfully causing a web application, database and/or web server to run with anomalous behavior. Each mutated packet was generated from one of nine markup tags with no repetition. Table 8-1 summarizes each experiment’s testing environment, testing descriptions (using which markup tag for mutation), and the outcome(s) associated with
the markup tag. The testing environment of experiments 1 to 5 and 7 to 9 were conducted on Windows 2000, the internet information services (Microsoft) server 5.0 and ASP 3.0. The sixth experiment was conducted on Ubuntu 8.04.3 and the web server Apache2.

<table>
<thead>
<tr>
<th>Markup Tags</th>
<th>Testing descriptions</th>
<th>Testing environments</th>
<th>Outcomes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;caseSensitive&gt;</td>
<td>change the HTTP method POST to post</td>
<td>Windows, IIS, ASP, Access</td>
<td>Normally processed the request message</td>
</tr>
<tr>
<td>2 &lt;charSpecific&gt;</td>
<td>Alter HTTP/1.1 to HTTP@1.1</td>
<td>Windows, IIS, ASP, Access</td>
<td>Server error message</td>
</tr>
<tr>
<td>3 &lt;dateSpecific&gt;</td>
<td>Violating the date limitation - Introducing an incorrect date format e.g 32/01/1980</td>
<td>Windows, IIS, ASP, Access</td>
<td>ODBC jet engine crashed</td>
</tr>
<tr>
<td>4 &lt;enumeratedLiteral&gt;</td>
<td>Executing the HTTP method TRACE</td>
<td>Windows, IIS, ASP, Access</td>
<td>Received response message</td>
</tr>
<tr>
<td>5 &lt;valueLimitation&gt;</td>
<td>Increase the student number characters from 123 to 123456</td>
<td>Windows, IIS, ASP, Access</td>
<td>ODBC jet engine crashed</td>
</tr>
<tr>
<td>6 &lt;stringSpecific&gt;</td>
<td>Command injection - ; finger</td>
<td>Ubuntu, Apache2,</td>
<td>Illegally obtain system Information</td>
</tr>
<tr>
<td>7 &lt;syntaxSpecific&gt;</td>
<td>Change the MIME header from Content-Length: 60 to 60</td>
<td>Windows, IIS, ASP, Access</td>
<td>Server error message</td>
</tr>
<tr>
<td>8 &lt;length&gt;</td>
<td>Breaking the semantic relationship between Content-Length and message body</td>
<td>Windows, IIS, ASP, Access</td>
<td>Wasting server resources</td>
</tr>
<tr>
<td>9 &lt;Legalchars&gt;</td>
<td>Input illegal characters into the form field</td>
<td>Windows, IIS, ASP, Access</td>
<td>ODBC jet engine crashed</td>
</tr>
</tbody>
</table>

Table 8-1: Nine experiment setup summary
**Vulnerability detected by caseSensitive tag**

This experiment uses caseSensitive tag to change the method of the request message “POST” to “post”. The purpose of this experiment was to test how the IIS server reacts with this unimplemented method. Surprisingly, the IIS accepted the request message and stored the posted message to the database. Based on the unexpected result, this experiment was retested by replacing the IIS server with the Apache2 server. The result of the replayed message was rejected by the Apache2 server. This experiment revealed that the implementation of the IIS server did not follow the specification. Table 8-2 shows the request packet and response packet of the IIS test run.

<table>
<thead>
<tr>
<th>Request Packet</th>
<th>Response Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>post /return.asp HTTP/1.1 Host: 192.168.0.5 User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11 Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,/<em>;q=0.5 Accept-Language: en-us,en;q=0.5 Accept-Encoding: gzip, deflate Accept-Charset: ISO-8859-1,utf-8;q=0.7,</em>;q=0.7 Keep-Alive: 300 Connection: keep-alive Referer: <a href="http://192.168.0.5/">http://192.168.0.5/</a> Content-Type: application/x-www-form-urlencoded Content-Length: 60 FirstName=John&amp;LastName=Smith&amp;SNumber=123&amp;DOB=01%2F01%2F1980</td>
<td>HTTP/1.1 200 OK Server: Microsoft-IIS/5.0 Date: Sun, 22 Nov 2009 19:08:10 GMT Content-Length: 265 Content-Type: text/html Set-Cookie: ASPSESSIONIDAASCSARQ=IIMLAGMALKCI ANDOGALFMEAN; path=/ Cache-control: private &lt;!DOCTYPE HTML PUBLIC &quot;-/W3C//DTD HTML 4.01 Transitional//EN&quot;&gt; &lt;html&gt; &lt;head&gt; &lt;title&gt;Student Info&lt;/title&gt; &lt;meta http-equiv=&quot;Content-Type&quot; content=&quot;text/html; charset=iso-8859-1&quot;&gt; &lt;/head&gt; &lt;body&gt; Data added to database successfully. &lt;/body&gt; &lt;/html&gt;</td>
</tr>
</tbody>
</table>

**Table 8-2: Vulnerability detected by caseSensitive tag**

**Vulnerability detected by charSpecific tag**

This experiment uses charSpecific tag to alter the forward slash character of HTTP/1.1 to HTTP@1.1. The purpose of this experiment was to test how the IIS handles
the packet with a syntax error. The testing result was as expected. The request message was rejected by the IIS server and the message “400 Bad Request” was received from the server. This experiment revealed that the implementation of the IIS server was able to handle this kind of attack. Table 8-3 shows the request packet and response packet of this test run.

<table>
<thead>
<tr>
<th>Request Packet</th>
<th>Response Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /return.asp HTTP/1.1</td>
<td>HTTP/1.1 400 Bad Request</td>
</tr>
<tr>
<td>Host: 192.168.0.5</td>
<td>Server: Microsoft-IIS/5.0</td>
</tr>
<tr>
<td>User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11</td>
<td>Date: Sun, 22 Nov 2009 19:17:01 GMT</td>
</tr>
<tr>
<td>Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,<em>/</em>;q=0.5 Accept-Language: en-us,en;q=0.5 Accept-Encoding: gzip,deflate Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7 Keep-Alive: 300 Connection: keep-alive Referer: <a href="http://192.168.0.5/">http://192.168.0.5/</a> Content-Type: application/x-www-form-urlencoded Content-Length: 60</td>
<td>Content-Type: text/html Content-Length: 87</td>
</tr>
<tr>
<td>FirstName=John&amp;LastName=Smith&amp;SNumber=123&amp;DOB=01%2F01%2F1980</td>
<td></td>
</tr>
</tbody>
</table>

Table 8-3: Vulnerability detected by charSpecific tag

**Vulnerability detected by dateSpecific tag**

This experiment uses dateSpecific tag to change the date from 01/01/1980 to 32/01/1980. The purpose of this experiment was to test how the database (Microsoft Access) handles exceeding the date limitation (greater than 31 days). The testing resulted in the crashing of the ODBC jet engine, generation of the error message “80040e07” and the IIS server returned the message “HTTP/1.1 500 Internal Server Error”. Table 8-4 shows the request packet and response packet of this test run.
### Table 8-4: Vulnerability detected by dateSpecific tag

<table>
<thead>
<tr>
<th>Request Packet</th>
<th>Response Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /return.asp HTTP/1.1</td>
<td>HTTP/1.1 100 Continue</td>
</tr>
<tr>
<td>Host: 192.168.0.5</td>
<td>Server: Microsoft-IIS/5.0</td>
</tr>
<tr>
<td>User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220</td>
<td>Date: Sun, 22 Nov 2009 19:23:14 GMT</td>
</tr>
<tr>
<td>BonEcho/2.0.0.11</td>
<td></td>
</tr>
<tr>
<td>Accept:</td>
<td></td>
</tr>
<tr>
<td>text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,plain;q=0.8,image/png,<em>/</em>;q=0.5</td>
<td></td>
</tr>
<tr>
<td>Accept-Language: en-us,en;q=0.5</td>
<td></td>
</tr>
<tr>
<td>Accept-Encoding: gzip, deflate</td>
<td></td>
</tr>
<tr>
<td>Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7</td>
<td></td>
</tr>
<tr>
<td>Keep-Alive: 300</td>
<td></td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td></td>
</tr>
<tr>
<td>Referer: <a href="http://192.168.0.5/">http://192.168.0.5/</a></td>
<td></td>
</tr>
<tr>
<td>Content-Type: application/x-www-form-urlencoded</td>
<td></td>
</tr>
<tr>
<td>Content-Length: 60</td>
<td></td>
</tr>
<tr>
<td>FirstName=John&amp;LastName=Smith&amp;SNumber=123&amp;DOB=3/2/1980</td>
<td></td>
</tr>
</tbody>
</table>

#### Vulnerability detected by enumeratedLiteral tag

This experiment uses enumeratedLiteral tag to change the method POST of the request message to TRACE. The purpose of this experiment was to test whether or not the method TRACE could be invoked. This method is not recommended to be available for users. It can be exploited for cross-site tracing attacks. This experiment revealed that the method TRACE was available at the default configuration of the IIS server. Table 8-5 shows the request packet and response packet of this test run.
Table 8-5: Vulnerability detected by enumeratedLiteral tag

**Vulnerability detected by valueLimitation tag**

This experiment uses valueLimitation tag to increase the value of a student number from 123 to 123456. The purpose of this experiment was to test how the database (Microsoft Access) handles the two-byte integer overflow. The testing result was the ODBC jet engine crashed and generated an error message “80040e57”. The IIS server returned the message “HTTP/1.1 500 Internal Server Error”. Table 8-6 shows the request packet and response packet of this test run.
Table 8-6: Vulnerability detected by valueLimitation tag

Vulnerability detected by stringSpecific tag

This experiment uses stringSpecific tag to inject the Linux system command “finger” to the web server in an attempt to steal system information. This attack successfully received user(s) logon information. It is because the CGI application program did not properly sanitize the user’s input string and allowed the system to invoke the piggyback command. Table 8-7 shows the request packet and response packet of this test run and the last two lines of the response packet is the user logon information caused by the system command “finger”.

<table>
<thead>
<tr>
<th>Request Packet</th>
<th>Response Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /return.asp HTTP/1.1</td>
<td>HTTP/1.1 100 Continue</td>
</tr>
<tr>
<td>Host: 192.168.0.5</td>
<td>Server: Microsoft-IIS/5.0</td>
</tr>
<tr>
<td>User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220</td>
<td>Date: Sun, 22 Nov 2009 19:49:26 GMT</td>
</tr>
<tr>
<td>BonEcho/2.0.0.11</td>
<td>HTTP/1.1 500 Internal Server Error</td>
</tr>
<tr>
<td>Keep-Alive: 300</td>
<td>Server: Microsoft-IIS/5.0</td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td>Date: Sun, 22 Nov 2009 19:49:26 GMT</td>
</tr>
<tr>
<td>Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,<em>/</em>;q=0.5</td>
<td>Content-Length: 255</td>
</tr>
<tr>
<td>Accept-Language: en-us,en;q=0.5</td>
<td>Content-Type: text/html</td>
</tr>
<tr>
<td>Accept-Encoding: gzip,deflate</td>
<td>Set-Cookie: ASPSESSIONIDAASCSARQ=MJMLAGMALDCF</td>
</tr>
<tr>
<td>Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7</td>
<td>EHEJIAKAFAME; path=/</td>
</tr>
<tr>
<td>Keep-Alive: 300</td>
<td>Cache-control: private</td>
</tr>
<tr>
<td>Referer: <a href="http://192.168.0.5/">http://192.168.0.5/</a></td>
<td>&lt;font face=&quot;Arial&quot; size=2&gt;</td>
</tr>
<tr>
<td>Content-Type: application/x-www-form-urlencoded</td>
<td>&lt;p&gt;Microsoft JET Database Engine&lt;/p&gt;</td>
</tr>
<tr>
<td>Content-Length: 60</td>
<td>&lt;font face=&quot;Arial&quot; size=2&gt;error '80040e57'&lt;/font&gt;</td>
</tr>
<tr>
<td>FirstName=John&amp;LastName=Smith&amp;SNumber=123456&amp;DOB=01%2F01%2F1980</td>
<td>&lt;p&gt;&lt;font face=&quot;Arial&quot; size=2&gt;Overflow&lt;/font&gt;&lt;/p&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;p&gt;&lt;font face=&quot;Arial&quot; size=2&gt;return.asp&lt;/font&gt;&lt;/p&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;font face=&quot;Arial&quot; size=2&gt;, line 11&lt;/font&gt;</td>
</tr>
</tbody>
</table>

Vulnerability detected by stringSpecific tag

This experiment uses stringSpecific tag to inject the Linux system command “finger” to the web server in an attempt to steal system information. This attack successfully received user(s) logon information. It is because the CGI application program did not properly sanitize the user’s input string and allowed the system to invoke the piggyback command. Table 8-7 shows the request packet and response packet of this test run and the last two lines of the response packet is the user logon information caused by the system command “finger”.

<table>
<thead>
<tr>
<th>Request Packet</th>
<th>Response Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /return.asp HTTP/1.1</td>
<td>HTTP/1.1 100 Continue</td>
</tr>
<tr>
<td>Host: 192.168.0.5</td>
<td>Server: Microsoft-IIS/5.0</td>
</tr>
<tr>
<td>User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220</td>
<td>Date: Sun, 22 Nov 2009 19:49:26 GMT</td>
</tr>
<tr>
<td>BonEcho/2.0.0.11</td>
<td>HTTP/1.1 500 Internal Server Error</td>
</tr>
<tr>
<td>Keep-Alive: 300</td>
<td>Server: Microsoft-IIS/5.0</td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td>Date: Sun, 22 Nov 2009 19:49:26 GMT</td>
</tr>
<tr>
<td>Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,<em>/</em>;q=0.5</td>
<td>Content-Length: 255</td>
</tr>
<tr>
<td>Accept-Language: en-us,en;q=0.5</td>
<td>Content-Type: text/html</td>
</tr>
<tr>
<td>Accept-Encoding: gzip,deflate</td>
<td>Set-Cookie: ASPSESSIONIDAASCSARQ=MJMLAGMALDCF</td>
</tr>
<tr>
<td>Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7</td>
<td>EHEJIAKAFAME; path=/</td>
</tr>
<tr>
<td>Keep-Alive: 300</td>
<td>Cache-control: private</td>
</tr>
<tr>
<td>Referer: <a href="http://192.168.0.5/">http://192.168.0.5/</a></td>
<td>&lt;font face=&quot;Arial&quot; size=2&gt;</td>
</tr>
<tr>
<td>Content-Type: application/x-www-form-urlencoded</td>
<td>&lt;p&gt;Microsoft JET Database Engine&lt;/p&gt;</td>
</tr>
<tr>
<td>Content-Length: 60</td>
<td>&lt;font face=&quot;Arial&quot; size=2&gt;error '80040e57'&lt;/font&gt;</td>
</tr>
<tr>
<td>FirstName=John&amp;LastName=Smith&amp;SNumber=123456&amp;DOB=01%2F01%2F1980</td>
<td>&lt;p&gt;&lt;font face=&quot;Arial&quot; size=2&gt;Overflow&lt;/font&gt;&lt;/p&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;p&gt;&lt;font face=&quot;Arial&quot; size=2&gt;return.asp&lt;/font&gt;&lt;/p&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;font face=&quot;Arial&quot; size=2&gt;, line 11&lt;/font&gt;</td>
</tr>
<tr>
<td>Request Packet</td>
<td>Response Packet</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GET /~benkam/cgi-bin/oto?Name=Otonabee+%3Bfinger</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>HTTP/1.1</td>
<td>Date: Mon, 23 Nov 2009 03:12:40 GMT</td>
</tr>
<tr>
<td>Host: cetus.ee.queensu.ca</td>
<td>Server: Apache/2.2.8 (Ubuntu) DAV/2 SVN/1.5.1</td>
</tr>
<tr>
<td>User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11</td>
<td>mod_python/3.3.1 Python/2.5.2 PHP/5.2.4-2ubuntu5.6 with Suhosin-Patch mod_scgi/1.12 mod_ssl/2.2.8 OpenSSL/0.9.8g</td>
</tr>
<tr>
<td>Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8 ,image/png,<em>/</em>;q=0.5</td>
<td>Keep-Alive: timeout=15, max=100 Connection: Keep-Alive</td>
</tr>
<tr>
<td>Accept-Language: en-us, en;q=0.5 Accept-Encoding: gzip, deflate Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7</td>
<td>Transfer-Encoding: chunked Content-Type: text/plain</td>
</tr>
<tr>
<td>KeepAlive: 300 Connection: keep-alive Referer: <a href="https://cetus.ee.queensu.ca/~benkam/">https://cetus.ee.queensu.ca/~benkam/</a></td>
<td></td>
</tr>
</tbody>
</table>

Table 8-7: Vulnerability detected by stringSpecific tag

**Vulnerability detected by syntaxSpecific tag**

This experiment uses syntaxSpecific tag to change the “Content-Length: 60” to “: 60”. This experiment tests how the IIS server handles a request packet with a mal-formed MIME header. The testing result was the IIS server rejected the request and sent back an error message “400 Bad Request”. Table 8-8 shows the request packet and response packet of this test run.
Vulnerability detected by length tag

This experiment uses length tag to change the value of Content-Length from 60 to 60000. The purpose of this experiment was to test how the IIS server handles idle network traffic. The testing result was the IIS server did not disconnect the network connection. The server kept waiting for the remaining bytes from the request client. This experiment revealed that increasing the content length size could launch a denial of service attack. Table 8-9 shows the request packet and response packet of this test run.
<table>
<thead>
<tr>
<th>Request Packet</th>
<th>Response Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /return.asp HTTP/1.1</td>
<td>No response because the server was waiting for the remaining bytes!</td>
</tr>
<tr>
<td>Host: 192.168.0.5</td>
<td></td>
</tr>
<tr>
<td>User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220</td>
<td></td>
</tr>
<tr>
<td>BonEcho/2.0.0.11</td>
<td></td>
</tr>
<tr>
<td>Accept:</td>
<td></td>
</tr>
<tr>
<td>text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,<em>/</em>;q=0.5</td>
<td></td>
</tr>
<tr>
<td>Accept-Language: en-us,en;q=0.5</td>
<td></td>
</tr>
<tr>
<td>Accept-Encoding: gzip,deflate</td>
<td></td>
</tr>
<tr>
<td>Accept-Charset: ISO-8859-1,utf-8;q=0.7,*,q=0.7</td>
<td></td>
</tr>
<tr>
<td>Keep-Alive: 300</td>
<td></td>
</tr>
<tr>
<td>Connection: keep-alive</td>
<td></td>
</tr>
<tr>
<td>Referer: <a href="http://192.168.0.5/">http://192.168.0.5/</a></td>
<td></td>
</tr>
<tr>
<td>Content-Type: application/x-www-form-urlencoded</td>
<td></td>
</tr>
<tr>
<td>Content-Length: 60000</td>
<td></td>
</tr>
<tr>
<td>FirstName=John&amp;LastName=Smith&amp;SNumber=123&amp;DOB=01%2F01%2F1980</td>
<td></td>
</tr>
</tbody>
</table>

Table 8-9: Vulnerability detected by length tag

**Vulnerability detected by LegalChars tag**

This experiment uses LegalChars tag to change the numeric value of a student number to an alphabetic characters string “abc”. The purpose of this experiment was to test how the database (Microsoft Access) handles inputting the wrong data type variable. The testing result was the ODBC jet engine crashed and generated an error message “80040e07”. The IIS server returned a message “HTTP/1.1 500 Internal Server Error”. Table 8-10 shows the request packet and response packet of this test run.
Table 8-10: Vulnerability detected by LegalChars tag

8.3 Vulnerability found on a commercial web application

The second set of experiments tests a commercial PHP image gallery web application [22]. According to their web site description, the major differences between the trial version and the paid version are not related to any security issues. Therefore, the security testing results of the trial version could reflect the paid version. Table 8-11 shows the setup information for this experiment.

Table 8-11: The second experimental setup

<table>
<thead>
<tr>
<th>Computer</th>
<th>Operating system</th>
<th>Memory</th>
<th>Web server</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plentinum II</td>
<td>Ubuntu 8.04</td>
<td>256M</td>
<td>Apache2</td>
<td>MySQL5</td>
</tr>
</tbody>
</table>

This testing contained a total of 1086 test cases. The injector sent test cases to the web server one by one and received a sequence of responses for each test case. After receiving a sequence of responses, the injector sent a SQL query to the database to check
whether or not a specially crafted script was successfully stored in the database. If the query returned a message was true, then the injector stopped the testing. Otherwise, the injector restored the database and started the next test-run. The injector stopped after running 4.81 hours of testing. As a result, the PacketsSentLog showed test case 1037 was the last test case, and the specially crafted script of this test case was found in the database. Table 8-12 shows the experimental data and Figure 8-1 shows that the specially crafted script replaced the original value of lastname “Kam”.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create 1086 Mutants</td>
<td>10.658s</td>
</tr>
<tr>
<td>Remove tags</td>
<td>49.476s</td>
</tr>
<tr>
<td>Injector runtime</td>
<td>17326.697s</td>
</tr>
<tr>
<td>Total</td>
<td>17386.831s</td>
</tr>
</tbody>
</table>

Table 8-12: Experiment two run time
Figure 8-1: The specially crafted script replaced the original value of lastname “Kam”.

```plaintext
POST /register.php HTTP/1.1
Host: 192.168.1.102
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.11) Gecko/20071220 BonEcho/2.0.0.11
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: gallarific_email=ben_kam%40hotmail.com; gallarific_name=Kam;
PHPSESSID=82d5b0a1a336932d6e10ae3f39a03c25a
Content-Type: multipart/form-data; boundary=---------------------------7928864061811355872564925266
Content-Length: 1200

-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="doregister"
true
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="username"
benkam
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="password"
123456
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="passwordc"
123456
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="email"
ben_kam@hotmail.com
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="firstname"
Ben
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="lastname"
<script>document.location.href="www.notarealsite.com";</script>
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="website"
http://192.168.1.10/~benkam
-----------------------------7928864061811355872564925266
Content-Disposition: form-data; name="photo"; filename=""
Content-Type: application/octet-stream
```
Figure 8-1 shows improper sanitizing of the input string. This reflects a vulnerability that allows the attacker to launch different kinds of attacks that are completely dependent on the specially crafted script.

8.4 Faultfinding on the open source KOrganizer

The third experimental set tested the open source KOrganizer [32]. KOrganizer is the scheduling component and the calendar of Kontact. Kontact is a personal information manager which can be integrated in KDE. KDE provides GUI interfaces for Linux and UNIX platforms to run applications.

iCalendar (iCal) is a computer format calendar file for internet users to send meeting requests to other users via email. Recipients may use email or iCal software, such as KOrganizer, to open the iCal file to retrieve the date and time of the meeting or counter propose another time for the meeting. iCal uses standards track RFC 2445 [15] text-based protocol for communication.

8.4.1 KOrganizer experimental setup

In order to use SST to test the KOrganizer, a few tiny changes of the framework is required to accommodate the testing needed. First, the Sniffer and the Decoder will not be used for this testing since these two components are protocol (HTTP) dependent components. Second, the travel agent will not be used since this testing only requires KOrganizer to open a mal-formed iCal file. Third, the oracle must be rewritten for this testing. This oracle is much simpler than the HTTP testing since the oracle only has to determine whether or not the KOrganizer has crashed after opening the iCal file. Fourth, a test driver is created to run the entire testing. Lastly, the iCal protocol specification has
to be built for creating test cases. Figure 8-2 depicts the essential components of this experiment.

In Figure 8-2, there are a total of five components for this testing. The first component is a test driver. The test driver is responsible for the whole testing run. The second component is the KOrganizer software. The third component is the xmacroplay. It is responsible for replaying a sequence of actions such as opening and closing the KOrganizer. The fourth component is the oracle. Lastly, a mutant directory contains all the test cases.

### 8.4.2 KOrganizer test driver

The test driver is a simple script that fetches a test case from the mutant directory to start testing. Referring to Figure 8-2, the work flow of the test driver is as follows. First, a test case is copied to the KOrganizer default directory “/home/benkam/.kde/share/apps/korganizer” and the mutant number is recorded to the test log. Second, the test driver fires up the KOrganizer software and invokes the xmacroplay to perform a sequence of actions to open the mal-formed iCal file. Third, the test driver instructs the
oracle to check whether or not the KOrganizer has crashed. If the KOrganizer has crashed, then an error will be reported to the error log. The test driver will repeat the same procedures for each test case. The testing will be finished when all the test cases have been tested. Figure 8-3 shows the complete code of the test driver.

```bash
#!/bin/bash
# kills all korganizer processes running in the background...
if [[ $1 == "kill" ]]; then
    for p in `ps uax | grep korganizer | grep -v grep | awk '{ print $2 }'`; do kill -9 $p; done
    exit;
fi

testdir="/home/benkam/mutants"
korgdir="/home/benkam/.kde/share/apps/korganizer"
errorlog="/home/benkam/Desktop/error.log"
iCalMutant="/home/benkam/Desktop/iCal-mutants.log"

rm -f $iCalMutant
echo "Starting..." >> $iCalMutant
rm -f $errorlog

for FILE in $testdir/*; do
    if [[ ! -f $FILE ]]; then
        continue;
    fi
    echo "Opening $FILE..." >> $iCalMutant
    rm -f $korgdir/iCalTest.ics
    cp $FILE $korgdir/iCalTest.ics
    rm -f /home/benkam/.kde/share/config/kresources/calendar/stdrc
    korganizer
    sleep 3
    cat testing.macro | xmacroplay -d 1000
    echo "resetting..."
    sleep 5
    if [[ ! -n `ps uax | grep -v grep | awk '{ print $11 }' | grep korganizer` ]]; then
        for p in `ps uax | grep korganizer | grep -v grep | awk '{ print $2 }'`; do
            echo "killing process $p"
            kill -9 $p;
        done
    fi
    echo $FILE >> $errorlog
done
```

Figure 8-3: The essential components for KOrganizer testing
8.4.3 The xmacroplay

The xmacroplay is a program that controls another application with a keystroke file. The keystroke file contains a sequence of keystroke or mouse click instructions. The xmacroplay will perform all the keystrokes and/or mouse clicks according to the keystroke file so as to simulate the user’s actions, such as open a file and close the application. Figure 8-4 shows the partial keystroke file. The xmacroplay is used to perform GUI testing of this experiment because no documentation can be found about the command line for KOrganizer in order to open the iCal file.

| MotionNotify 17 0 |
| ButtonPress 1 |
| ButtonRelease 1 |
| MotionNotify 153 145 |
| ButtonPress 1 |
| ButtonRelease 1 |
| MotionNotify 334 155 |
| ButtonPress 1 |
| ButtonRelease 1 |
| MotionNotify 495 340 |
| ButtonPress 1 |
| ButtonRelease 1 |

Figure 8-4: The partial keystroke file

8.4.4 The Oracle

The oracle of this experiment is very simple. It uses a grep command to check whether or not the KOrganizer is still running in the background, five seconds after being closed by the xmacroplay. If the KOrganizer processes still exist, then it means the KOrganizer has crashed and cannot be closed normally by the xmacroplay. As a result, a fault is considered to have been found.
8.4.5 Test cases generation

The procedures of test cases generation are exactly the same as the HTTP testing.

The test cases are the mutants of the original iCal file. Figure 8-5 shows the original well-formed iCal file.

```
BEGIN:VCALENDAR
PRODID:-//K Desktop Environment//NONSGML KOrganizer 4.1.4//EN
VERSION:2.0
BEGIN:VEVENT
DTSTAMP:20090828T070531Z
ORGANIZER;CN="Thomas Dean";MAILTO:tom.dean@queensu.ca
ATTENDEE;RSVP=TRUE;PARTSTAT=NEEDS-ACTION;ROLE=REQ-PARTICIPANT:mailto:
    Kam@cs.queensu.ca
ATTENDEE;CN="Thomas Dean";RSVP=FALSE;PARTSTAT=ACCEPTED;
    ROLE=REQ-PARTICIPANT:mailto:tom.dean@queensu.ca
CREATED:20090721T185818Z
UID:13EF25D5-8DDC-46C6-86B4-E164335FFE08
SEQUENCE:4
LAST-MODIFIED:20090828T070446Z
DESCRIPTION:
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
SUMMARY:ben
DURATION:PT1H
DTSTART;TZID=Canada/Eastern:20090730T143000
TRANSP:OPAQUE
END:VEVENT
BEGIN:VTIMEZONE
TZID:Canada/Eastern
BEGIN:DAYLIGHT
TZOFFSETFROM:-0500
TZOFFSETTO:-0400
DTSTART:20070311T020000
RRULE:FREQ=YEARLY;BYDAY=2SU;BYMONTH=3
TZNAME:EDT
END:DAYLIGHT
BEGIN:STANDARD
TZOFFSETFROM:-0400
TZOFFSETTO:-0500
DTSTART:20071104T020000
RRULE:FREQ=YEARLY;BYDAY=1SU;BYMONTH=11
TZNAME:EST
END:STANDARD
END:VTIMEZONE
END:VCALENDAR
```

Figure 8-5: The original well-formed iCal file
Figures 8-6 and 8-7 show the partial iCal generalized grammar and the partial iCal low level description file, respectively.

```
define program
    [repeat iCalObject]
end define

define iCalObject
    
'BEGIN ': 'VCALENDAR [CRLF] [iCalBody] 'END ': 'VCALENDAR [CRLF]
end define

define iCalBody
    [repeat calprops] [repeat component]
end define ...
define dtstamp
    'DTSTAMP [repeat stmparam] ': [date_time] [CRLF]
end define

define stmparam
    '; [xparam]
end define ...
```

Figure 8-6: The partial iCal generalized grammar

```
define dtstamp
    'DTSTAMP [repeat stmparam] ': <dateSpecific> [date_time] </dateSpecific> [CRLF]
end define
```

Figure 8-7: The partial iCal low level description language

In this experiment, the caseSensitive tag, charSpecific tag, dateSpecific tag, syntaxSpecific tag, valueLimitation tag, and the stringSpecific tag were used to generate test cases. Figure 8-8 and Figure 8-9 illustrate the partial iCal.lproto and the partial mutated iCal file (one of the test cases by dateSpecific tag). Note that a few stringSpecific test cases will be generated by using shell scripts, for example, the original string will be replaced by a huge string with 16M bytes size.
In Figure 79, the DTSTAMP was changed from the original 20090828T070531Z to DTSTAMP:999999999999999.

BEGIN:VCALENDAR
PRODID:-//K Desktop Environment//NONSGML KOrganizer 4.1.4//EN
VERSION:2.0
BEGIN:VEVENT
DTSTAMP:999999999999999
ORGANIZER;CN="Thomas Dean":MAILTO:tom.dean@queensu.ca
ATTENDEE;RSVP=TRUE;PARTSTAT=NEEDS-ACTION;ROLE=REQ-PARTICIPANT:mailto:
Kam@cs.queensu.ca
ATTENDEE;CN="Thomas Dean",RSVP=FALSE;PARTSTAT=ACCEPTED;
ROLE=REQ-PARTICIPANT:mailto:tom.dean@queensu.ca
...

8.4.6 Testing results

The test driver copied a test case to the KOrganizer default directory, opened the KOrganizer, instructed xmacroplay to perform a sequence of actions to open the test case and checked if the KOrganizer has crashed. The test driver repeated these steps 1026
times because there were a total of 1026 test cases. The total running time was 244188 seconds, or 67.83 hours. The reasons for taking 67.83 hours to complete the KOrganizer are as follow. First of all, we could not find the command that can instruct the KOrganizer to import and open the iCal file. Second, all the testing processes used xmacroplay to perform a sequence of mouse clicks so as to replay a tester’s action to open the KOrganizer to import and open the iCal file. In order to ensure the xmacroplay successfully replays this process, we had to put some threads to sleep for several seconds before they start to run. Otherwise, the xmacroplay will fail to play the mouse click instruction sequence. For example, the coordinates of clicking the “import” option of the File pull down menu are “MotionNotify 153 145”; therefore, we have to delay the xmacroplay to execute the instructions until the KOrganizer has completely rendered the screen. Otherwise, the xmacroplay may click the wrong place and fail the instruction sequence. Table 8-13 and Table 8-14 show the experimental setup information and testing data, respectively.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Operating system</th>
<th>Memory</th>
<th>KOrganizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD3300+</td>
<td>Ubuntu 8.10</td>
<td>512M</td>
<td>4.1.4</td>
</tr>
</tbody>
</table>

Table 8-13: The third experimental setup information

<table>
<thead>
<tr>
<th>Create 1086 Mutants</th>
<th>9.069s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove tags</td>
<td>45.742s</td>
</tr>
<tr>
<td>Test driver runtime</td>
<td>244188s</td>
</tr>
<tr>
<td>Total</td>
<td>244242.811s</td>
</tr>
</tbody>
</table>

Table 8-14: The data from experiment three

The test log file and error log file were generated after the test driver has stopped running. The test log contained a total of 1026 file names which indicated that all the
mutants had been tested. The error log contained a mutant number iCal-559 which was a stringSpecific mutant and the original content of the DESCRIPTION: AAAAA…… was replaced with a giant string (16M bytes size). A further step to verify this mal-formed iCal-559 file can cause the KOrganizer to crash is to manually import the iCal file to the KOrganizer and reopen it. The result was the same; the KOrganizer crashed and caused the signal 11 (SIGSEGV).

8.5 Mimicking PROTOS Test-Suite: co-http-reply

The primary goal of mimicking PROTOS test suite: co-http-reply is to show that SST can conduct the same testing as other text-based testing tools. To compare the performance and effectiveness with PROTOS is beyond the scope of this research. Besides, would the testing result be the same if the same test cases are used to attack the same server? It will be shown that SST can generate the same test cases (or even more) as co-http-reply. Therefore, the co-http-reply test cases will be the subset of the SST test cases.

In this section, SST demonstrates its flexibilities to mimick the venerable network PROTOS Test-Suite: co-http-reply [49] developed by University of Oulu. This test suite contains a total of 3966 test cases. After analyzing these test cases, several patterns to mutate the original response packet were identified. Table 8-15 shows all the identified patterns.

In Table 8-15, if the target replacement is a character or a sequence of characters, then the replacement pattern can consist of different character or a different sequence of characters. The replacement pattern can also consist of a different character or a different
sequence of characters with a leading white space or a negative value sign. Once the mutation patterns have been identified, using SST to generate test cases equivalent to co-http-reply could be very easy. First of all, a HTTP response generalized grammar has to be created. Figure 8-10 depicts the partial HTTP response generalized grammar.

<table>
<thead>
<tr>
<th>Targets</th>
<th>Examples</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>String, value, separator</td>
<td>HTTP, ok, 1.1, 200, a white space, /, ;</td>
<td>Replace with nothing, negative sign, a white space, a binary character</td>
</tr>
<tr>
<td>Replace with a character</td>
<td>1 character</td>
<td>1 character</td>
</tr>
<tr>
<td></td>
<td>17 characters</td>
<td>17 characters</td>
</tr>
<tr>
<td></td>
<td>68 characters</td>
<td>68 characters</td>
</tr>
<tr>
<td></td>
<td>272 characters</td>
<td>272 characters</td>
</tr>
<tr>
<td></td>
<td>1088 characters</td>
<td>1088 characters</td>
</tr>
<tr>
<td></td>
<td>4352 characters</td>
<td>4352 characters</td>
</tr>
<tr>
<td></td>
<td>17408 characters</td>
<td>17408 characters</td>
</tr>
<tr>
<td></td>
<td>69632 characters</td>
<td>69632 characters</td>
</tr>
<tr>
<td>Replace with %s</td>
<td>1 %s</td>
<td>1 %s</td>
</tr>
<tr>
<td></td>
<td>17 %s</td>
<td>17 %s</td>
</tr>
<tr>
<td></td>
<td>68 %s</td>
<td>68 %s</td>
</tr>
<tr>
<td></td>
<td>272 %s</td>
<td>272 %s</td>
</tr>
<tr>
<td></td>
<td>1088 %s</td>
<td>1088 %s</td>
</tr>
<tr>
<td>Replace with %n</td>
<td>1 %n</td>
<td>1 %n</td>
</tr>
<tr>
<td></td>
<td>17 %n</td>
<td>17 %n</td>
</tr>
<tr>
<td></td>
<td>68 %n</td>
<td>68 %n</td>
</tr>
<tr>
<td></td>
<td>272 %n</td>
<td>272 %n</td>
</tr>
<tr>
<td></td>
<td>1088 %n</td>
<td>1088 %n</td>
</tr>
</tbody>
</table>
| Replace with %2048d |                                | Replace with %2048d|%
| Replace with %4096d |                                | Replace with %4096d|%
| Replace with %8192d |                                | Replace with %8192d|%
| Replace with %2048d%%n |                            | Replace with %2048d%%n|
| Replace with %4096d%%n |                            | Replace with %4096d%%n|
| Replace with %8192d%%n |                            | Replace with %8192d%%n|

Table 8-15: PROTOS Test-Suite: co-http-reply test cases characteristics

The next step is constructing a low level description language file to decide what parts should be mutated. Lastly, using markup tags charSpecific and stringSpecific should be sufficient to generate all the test cases. Figure 8-11 represents the original response packet from co-http-reply. For demonstration purposes, let’s say the targets for mutation are the “HTTP” and the “/” from the first line of the Figure 8-11. Therefore, the low level description file should have the HTTP non-terminal and the slash non-terminal marked up. Since the SST will use libraries to generate test cases for these two tags;
therefore, the Character and the String files have to fill in all the designed patterns to add to the respective libraries.

```plaintext
define program
  [response_message]
end define

define response_message
  [response_message_no_content]
  | [status_line] [repeat headers_messages] [CRLF] [opt message_body]
end define

define response_message_no_content
  [status_line] [repeat headers_messages_no_content] [CRLF]
end define

define status_line
  [http_version] [space] [status_code] [space] [reason_phrase] [CRLF]
end define

define http_version
  [id] [slash] [number]
end define

define slash
  '/
end define
...
```

Figure 8-10: The partial HTTP Response generalized grammar

```
HTTP/1.1 200 OK
Server: Bugbear/020
Date: Monday, 03-Jan-02 12:12:12 GMT
Vary: *
Connection: close
Last-Modified: Mon, 12 Jan 1970 15:46:40 EET
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 80

<html><body><h1>c05-http-reply</h1>  <p>This test-case is 1</p></body></html>
```

Figure 8-11: The original HTTP response packet from co-http-reply

Figure 8-12 illustrates the low level description file. The replacement patterns between charSpecific tag and stringSpecific tag are pretty much the same. Basically, using one markup tag (stringSpecific) should create all mutants. However, to clarify for
the demonstration, two tags are used. Figure 8-13 shows the partial content of the String file.

```plaintext
define http_version
    <stringSpecific> [id] </stringSpecific> <charSpecific> [slash] </charSpecific> [number]
end define
```

Figure 8-12: The partial HTTP Response protocol specification

Once all the replacement patterns have been documented and added to the libraries, test cases are ready to be generated. The purpose of this mimicking is to show that whatever the existing text-based communication protocols testing tool can do, so can SST. Because SST can generate the same amount and quality of test cases as co-http-reply testing (greater amounts depending on the placement of the markup tags); therefore, SST can be considered to subsume the other text-based protocol techniques.
8.6 Chapter Conclusion

The capabilities and flexibilities of SST have been demonstrated in this chapter. A successful attack on a commercial web site and the crashing of the open source KOrganizer show the effectiveness of SST. Showing SST can also perform the same testing as the other testing tools is further proof of the SST capabilities and flexibilities.
Chapter 9

Conclusion and Future Work

9.1 Conclusion

The contributions of this thesis –

- Creation of a light weight security testing framework usable for most text-based communication protocols.
- Extension mechanism for easily adding new markup and mutators.
- Integration with external mutators for embedded binary data.
- We have demonstrated the framework with attacks on HTTP and iCalendar application.

The advantage of the SST testing framework is that it is capable of testing protocol applications and the system. For example, a new web site will be hosted by a particular web server. The assumption is that the conformance testing of the web site has been done and that the web server is problem-free. However, the web site still cannot go live, especially for high risk businesses. First, it is necessary to conduct security testing of the web site to ensure that no known tricks are capable of breaking the applications. Second, joining the web applications to the web server may introduce some security issues. The web applications and web server are two different modules; their communication completely relies on data passing between the two layers. The interpretation of the same data may not be the same for different layers, which may yield different results (lots of security faults can be fixed by sanitizing the input data). We have
to put them together as one system to perform security testing. SST is an effective and efficient security testing tools and can test these two modules at the same time.

Although we have not yet explored all possible markup tags, the effectiveness of SST is still shown in this research. All in all, we have just seen the tip of the iceberg of SST. In fact SST is not for any particular text-based communication protocol. It can also test all existing and up-and-coming future text-based protocols. Continuing to explore more markup tags can increase the chance of revealing faults. Optimizing the performance and effectiveness of SST will help software developers tremendously by saving the time and effort of performing tedious and time consuming testing tasks. The creation of an intelligent user interface can not only reduce testers' learning curve but also eliminate the process of preparing the testing protocol grammar. As a result, testers without any protocol grammar knowledge are still able to use SST. SST has the potential to hold a great deal of academic and industrial research value. SST could become a de facto standard and recommended security testing tool in the future.

9.2 Future Work

SST seems to be robust; however, there are some issues that must be addressed. First, the SST framework lacks a sophisticated oracle. Second, the types of markup tags can be expanded. Third, creating an intelligent user interface may help to drastically reduce the user learning curve. And last but not least, the source code of SST has to be unified to allow easy maintenance. Each of these issues will be discussed in this section.
9.2.1 Modifying the oracle

In order to have a fully automated testing system, an oracle plays an important role. An ideal oracle should be able to generate a final report stating which test cases successfully revealed faults and also provide a detailed log so that testers can further investigate the details. The current version oracle used in SST is not a perfect oracle. The oracle cannot fulfill the above description all the time. Determining whether or not a well-formed response message is received from the web server is a complex task.

The difficulty of the comparison of the two responses stems from the nature of the HTTP response. One criterion for a well-formed response is both response messages are identical. In addition, two different response messages can be considered well-formed if the response is legitimate. For example, the response message may contain the current time. Another example, if the method GET is changed to GOT e.g. GOT /hello.html HTTP/1.1 and sent to the server, should the oracle expect the response message “400 Bad Request” or “501 Method Not Implemented”? – both are legitimate responses. Besides, different web servers may have different response messages. Most importantly, both response messages represent that the server can handle this mutant and reject the request. This reflects a healthy server. For a thorough comparison, a sophisticated oracle would need to include all possible responses so as not to conclude falsely that the response message is illegitimate. Creating an oracle for this testing would be interesting empirical research.
9.2.2 Exploring the markup tags

First, the role of the markup tags is to find different vulnerabilities. The present markup tags focus mainly on breaking the constraints of the specification of the protocol grammar. SST does not have any markup tags that aim to break the constraints from its implementation language. For example, the Apache HTTP server is implemented by C language. If the markup tags also aimed to break the C language constraints, it may increase the chance of finding faults.

Second, only the relation tag and application level markup tags can have attributes sent to the mutation engines. In order to send attributes of other markup tags to the mutation engines, they must be modified. This modification can increase the flexibility for testers who want to test certain cases.

And last but not least, the system state information can also be embedded in the markup tag. Therefore, SST does not need the real time update file. This may increase the performance of the testing.

9.2.3 Implementing an intelligent GUI user interface

An initial design of an intelligent GUI interface will minimize the learning curve for testers. For example, if a tester wants to use a charSpecific markup tag to mutate the forward slash of “HTTP/1.1”, then the tester only needs to check the corresponding box on the GUI user interface. Basically, the tester does not need to know anything about markup tags or grammar.
Appendix A: Command-line commands

A1: To generate the httpRequest.lproto.

```
txl -idchars '-' MarkTagDescriptionFile genLowLevelLproto.txl - - grammar
httpRequest.grm
```

```
txl -idchars '-' httpRequest.lproto prettyprint.txl -o httpRequest.lproto
```

A2: To generate the optAndTag.grm

```
txl -txl -idchars '-' httpRequest.lproto buildGrammar.txl -o tagHttpRequest.grm - -tags
tagDictionary
```

```
txl -idchars '-' tagHttpRequest.grm prettyprint.txl -o tagHttpRequest.grm
```

```
cat optTagHttpRequest.grm tagHttpRequest.grm > optAndTag.grm
```

A3: To generate the InsertMarkup.txl

```
txl -raw optAndTag.grm buildMarkup.txl -o InsertMarkup.txl - tagHttpRequest.grm
optTagHttpRequest.grm
```

A4: To insert markup tags onto the login packet

```
./txl –txl descriptionLanguage genClassifiedPacket.txl
./doFindPacket PacketDirectory LoginPacket
./txl descriptionLanguage genMarkUpPacket.txl - -packet Login
./txl dropTagsAtLoginPacket.txl prettyPrintDropTag.txl - -basefile
dropTagsAtLoginPacket.txl
./txl Req00055.dat dropTagsAtLoginPacket.txl - -basefile Req00055.dat
```
References


workshop on Testing, analysis, and verification of web services and applications, pp.1 – 9, 2006.


