A Model-Driven Penetration Test Framework for Web Applications

Pulei Xiong

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the Ph.D. degree in Computer Science

University of Ottawa

Ottawa, Ontario, Canada

January 1, 2012

Abstract

Penetration testing is widely used in industry as a test method for web application security assessment. However, penetration testing is often performed late in a software development life cycle as an isolated task and usually requires specialized security experts. There is no well-defined test framework providing guidance and support to general testers who usually do not have in-depth security expertise to perform a systematic and cost-efficient penetration test campaign throughout a security-oriented software development life cycle.

In this thesis, we propose a model-driven penetration test framework for web applications that consists of a penetration test methodology, a grey-box test architecture, a web security knowledgebase, a test campaign model, and a knowledge-based PenTest workbench. The test framework enables general testers to perform a penetration test campaign in a model-driven approach that is fully integrated into a security-oriented software development life cycle. Security experts are still required to build up and maintain a web security knowledgebase for test campaigns, but the general testers are capable of developing and executing penetration test campaigns with reduced complexity and increased reusability in a systematic and cost-efficient approach.

A prototype of the framework has been implemented and applied to three web applications: the benchmark WebGoat web application, a hospital adverse event management system (AEMS), and a palliative pain and symptom management system (PAL-IS). An evaluation of the test framework prototype based on the case studies indicates the potential of the proposed test framework to improve how penetration test campaigns are performed and integrated into a security-oriented software development life cycle.

Acknowledgements

I would like to express my appreciation and gratitude to my supervisor Professor Liam Peyton for his guidance, encouragement and financial support throughout my thesis research.

My deepest love and respect to my wife and my parents for their encouragement and selfless support to help me achieve this.

Table of Contents

ABSTRACT .... **I**

ACKNOWLEDGEMENTS .... **III**

TABLE OF CONTENTS .... **IV**

LIST OF FIGURES .... **IX**

LIST OF TABLES .... **X**

LIST OF ACRONYMS .... **XI**

CHAPTER 1. INTRODUCTION **1**

1.1. PROBLEM STATEMENT 1

1.2. THESIS MOTIVATION 3

1.3. THESIS CONTRIBUTIONS 4

1.4. THESIS METHODOLOGY AND LIMITATIONS 5

1.5. SECURITY TERMS USED IN THIS THESIS 9

CHAPTER 2. BACKGROUND **12**

2.1. WEB APPLICATION ARCHITECTURE AND TECHNOLOGIES 12

2.1.1 Web Application Open Architecture *12*

2.1.2 Fundamental Web Technologies *15*

2.1.3 Web Application Functionality *22*

2.2. WEB APPLICATION SECURITY 23

2.2.1 Core Defense Mechanisms *23*

2.2.2 Basic Attack Categories and Common Vulnerabilities *26*

2.3. SOFTWARE SECURITY ENGINEERING 30

2.3.1 Software Security Development Life Cycle *31*

2.3.2 Software Quality Assurance *32*

2.3.3 Software Security Testing *32*

2.4. MODEL-DRIVEN SOFTWARE TESTING 33

2.4.1 Software Testing Fundamentals *33*

2.4.2 Model-Based Testing *35*

2.4.3 Model Driven Engineering *35*

2.4.4 Model-Driven Testing *36*

2.5. WEB SECURITY KNOWLEDGEBASE 37

2.5.1 Web Application Security Knowledge Sources *37*

2.5.2 Security Knowledge Representation *38*

CHAPTER 3. RELATED RESEARCH AND INDUSTRIAL PRACTICE 39

3.1. MICROSOFT SECURITY DEVELOPMENT LIFECYCLE 39

3.2. THE COMMON CRITERIA EVALUATION 40

3.3. WEB APPLICATION PENETRATION TESTING 42

3.3.1 Best Practices in Industry *43*

3.3.2 Expert Security Assessment *46*

3.3.3 Tool Support *49*

3.4. OTHER SECURITY ASSURANCE APPROACHES 50

CHAPTER 4. MODEL-DRIVEN PENETRATION TEST FRAMEWORK 52

4.1. PROBLEM DEFINITION 52

4.2. EVALUATION CRITERIA 56

4.3. TEST FRAMEWORK OVERVIEW 59

4.4. PENETRATION TEST METHODOLOGY 61

4.4.1 Overview *61*

4.4.2 Roles and Objectives *64*

4.4.3 Test Analysis *65*

4.4.4 Test Design *68*

4.4.5 Test Execution *73*

4.4.6 Application Maintenance *77*

4.5. GREY-BOX TEST ARCHITECTURE 78

4.5.1 Web Application Architecture 79

4.5.2 Entry Point 80

4.5.3 Check Point 82

4.6. WEB SECURITY KNOWLEDGEBASE 84

4.6.1 Attack Anatomy 84

4.6.2 Characteristics of Attack Vectors 85

4.6.3 Characteristics of Fuzz Vectors 86

4.6.4 Penetration Test Security Model 87

4.7. PENETRATION TEST CAMPAIGN MODEL 89

4.8. KNOWLEDGE-BASED PENTEST WORKBENCH 92

CHAPTER 5. TEST CAMPAIGN CASE STUDIES 94

5.1. OVERVIEW OF TEST CAMPAIGNS 94

5.1.1 WebGoat Test Campaign 94

5.1.2 AEMS Test Campaign 95

5.1.3 PAL-IS Test Campaign 96

5.1.4 The Roles of the Thesis Researcher in the Case Studies 98

5.2. TEST FRAMEWORK PROTOTYPE 99

5.2.1 Test Artifacts and Documents 100

5.2.2 Test Campaign Model 100

5.2.3 Knowledge-Based PenTest Workbench Implementation 102

5.2.4 Web Security Knowledgebase 106

5.3. WEBGOAT TEST CAMPAIGN 108

5.3.1 Introduction 108

5.3.2 Test Requirement Analysis 108

5.3.3 Test Case Design 109

5.3.4 Test Execution 112

5.3.5 Test Result Summary 113

5.4. AEMS TEST CAMPAIGN 114

5.4.1 Introduction 114

5.4.2 Test Requirement Analysis 115

5.4.3 Test Case Design 115

5.4.4 Test Execution 118

5.4.5 Test Result Summary 120

5.4.6 The Refinement of the Test Framework 121

5.5. PAL-IS TEST CAMPAIGN 122

5.5.1 Introduction 122

5.5.2 Test Requirement Analysis 122

5.5.3 Test Case Design 124

5.5.4 Test Execution 128

5.5.5 Test Result Summary 136

CHAPTER 6. EVALUATION 137

6.1. INTENDED USE OF FRAMEWORK AND CASE STUDIES 138

6.2. COMPARE WITH THE MS SDL 140

6.3. COMPARISON OF SECURITY ASSURANCE APPROACHES 149

6.4. COMPLEMENTARY NATURE OF THE PROPOSED TEST FRAMEWORK 154

6.4.1 Supplement to the Microsoft Secure Development Lifecycle 154

6.4.2 Use Vulnerability Scanner in the Test Framework 155

6.5. WEBGOAT REFERENCE ANALYSIS AND PERFORMANCE 156

6.6. LIMITATIONS 158

6.6.1 Design-Oriented Research Methodology 158

6.6.2 Software Development Life Cycle Model 159

6.6.3 Web Application Architecture 159

6.6.4 Simple Web Security Knowledgebase 160

6.6.5 Limited Scope of the Case Studies 160

CHAPTER 7. CONCLUSIONS **163**

7.1. SUMMARY OF CONTRIBUTIONS 163

7.2. FUTURE WORK 165

7.2.1 Build Up More Sophisticated Web Security Knowledgebase 165

7.2.2 Conduct Case Study in Larger Scope 166

7.2.3 Develop a Model-Drive Test Architecture 167

7.2.4 Extend the Proposed Test Framework 167

BIBLIOGRAPHY **169**

APPENDIX A1 WEBGOAT TEST ANALYSIS DOCUMENT **176**

APPENDIX A2 WEBGOAT TEST DESIGN DOCUMENT **177**

APPENDIX A3 WEBGOAT TEST EXECUTION DOCUMENT **192**

APPENDIX B1 AEMS TEST ANALYSIS DOCUMENT **193**

APPENDIX B2 AEMS TEST DESIGN DOCUMENTS **194**

APPENDIX B3 AEMS TEST EXECUTION DOCUMENT **205**

APPENDIX C1 PAL-IS TEST ANALYSIS DOCUMENT **207**

APPENDIX C2 PAL-IS TEST DESIGN DOCUMENT **209**

APPENDIX C3 PAL-IS TEST EXECUTION DOCUMENT **222**

List of Figures

FIGURE 1 HTTP URL....16
FIGURE 2 SECURE SOFTWARE DEVELOPMENT PROCESS MODEL AT MICROSOFT 39
FIGURE 3 PENETRATION TEST DEVELOPMENT – CURRENT PRACTICE 53
FIGURE 4 MODEL-DRIVEN PENETRATION TESTING....55
FIGURE 5 TEST FRAMEWORK OVERVIEW 61
FIGURE 6 ITERATIVE SOFTWARE DEVELOPMENT & TEST PROCESS....62
FIGURE 7 METHODOLOGY: TEST ANALYSIS 66
FIGURE 8 TEST ANALYSIS DOCUMENT TEMPLATE 67
FIGURE 9 METHODOLOGY: TEST DESIGN 69
FIGURE 10 TEST DESIGN DOCUMENT TEMPLATE 72
FIGURE 11 METHODOLOGY: TEST EXECUTION....73
FIGURE 12 TEST EXECUTION DOCUMENT TEMPLATE....76
FIGURE 13 AJAX WEB APPLICATION ARCHITECTURE 79
FIGURE 14 WEB APPLICATION PENETRATION TEST SECURITY MODEL 87
FIGURE 15 PENETRATION TEST CAMPAIGN MODEL 90
FIGURE 16 A KNOWLEDGE-BASED PENTEST WORKBENCH 93
FIGURE 17 TEST CAMPAIGN MODEL 101
FIGURE 18 TEST ARCHITECTURE AND THE KNOWLEDGE-BASED PENTEST WORKBENCH 103
FIGURE 19 A SAMPLE XSS VULNERABILITY....106
FIGURE 20 A SAMPLE ATTACK VECTOR FOR XSS 106
FIGURE 21 A SAMPLE FUZZ VECTOR FOR XSS....107
FIGURE 22 WEBGOAT APPLICATION FOOTPRINT 109
FIGURE 23 A SAMPLE USER SCENARIO IN THE WEBGOAT TEST CAMPAIGN 109
FIGURE 24 USE WEBSCARAB TO IDENTIFY ENTRY POINT 110
FIGURE 25 A SAMPLE ENTRY POINT IN THE WEBGOAT TEST CAMPAIGN 111
FIGURE 26 SAMPLE ATTACKS IN THE WEBGOAT TEST CAMPAIGN 111
FIGURE 27 A SAMPLE CHECK POINT IN THE WEBGOAT TEST CAMPAIGN 112
FIGURE 28 AEMS APPLICATION FOOTPRINT 115
FIGURE 29 A SAMPLE USER SCENARIO IN THE AEMS TEST CAMPAIGN 115
FIGURE 30 A SAMPLE ENTRY POINT IN THE AEMS TEST CAMPAIGN 116
FIGURE 31 SAMPLE ATTACKS IN THE AEMS TEST CAMPAIGN 117
FIGURE 32 A SAMPLE CHECK POINT IN THE AEMS TEST CAMPAIGN 118
FIGURE 33 "VALIDATOR" CODE SNIPPET IN AEMS 119
FIGURE 34 PAL-IS APPLICATION FOOTPRINT 123
FIGURE 35 PAL-IS FUNCTIONALITIES....123
FIGURE 36 A SAMPLE PAL-IS FUNCTIONALITY AND ITS POTENTIAL VULNERABILITY....124
FIGURE 37 A SAMPLE ENTRY POINT IN THE PAL-IS TEST CAMPAIGN....124
FIGURE 38 SAMPLE ATTACKS IN THE PAL-IS TEST CAMPAIGN 125
FIGURE 39 SAMPLE CHECK POINTS IN THE PAL-IS TEST CAMPAIGN....126
FIGURE 40 A SAMPLE TEST CASE IN THE PAL-IS TEST CAMPAIGN 127
FIGURE 41 SCREENSHOT OF THE ERROR MESSAGE FOR AN XSS ATTACK IN THE PAL-IS TEST CAMPAIGN 130
FIGURE 42 SCREENSHOT OF THE APPLICATION LOGS IN THE PAL-IS TEST CAMPAIGN....131
FIGURE 43 SCREENSHOT OF A SUCCESSFUL XSS ATTACK IN THE PAL-IS TEST CAMPAIGN....132
FIGURE 44 A SAMPLE PAL-IS TEST REPORT 135

List of Tables

TABLE 1 COMPARISON OF ENTRY POINTS 81
TABLE 2 COMPARISON OF CHECK POINTS 83
TABLE 3 THE WEBGOAT TEST CAMPAIGN SUMMARY 113
TABLE 4 THE WEBGOAT TEST RESULT SUMMARY 114
TABLE 5 THE AEMS TEST CAMPAIGN SUMMARY 121
TABLE 6 THE PAL-IS TEST CAMPAIGN SUMMARY 136
TABLE 7 COMPARISON WITH THE MS SDL 141
TABLE 8 COMPARISON OF SECURITY ASSURANCE APPROACHES 150
TABLE 9 WEBGOAT REFERENCE ANALYSIS AND PERFORMANCE 158

List of Acronyms

Acronym	Definition
AJAX	Asynchronous JavaScript and XML
AEMS	Adverse Event Management System
API	Application programming interface
ASCII	American Standard Code for Information Interchange
AVDL	Application Vulnerability Description Language
CAPEC	Common Attack Pattern Enumeration and Classification
CC	Common Criteria
CCRA	Common Criteria Recognition Arrangement
CCS	Common Criteria Schema
CEM	Common Evaluation Methodology
CSRF	Cross-Site Request Forgeries
CSS	Cascading Style Sheets
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DBMS	Database Management System
DOM	Document Object Model
DTD	Document Type Definition
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	HyperText Transfer Protocol

HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISECOM	The Institute for Security and Open Methodologies
JSON	JavaScript Object Notation
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MS SDL	Microsoft Security Development Lifecycle
NVD	National Vulnerability Database
OS	Operation System
OSSTMM	Open Source Security Testing Methodology Manual
OSVDB	The Open Source Vulnerability Database
OVAL	Open Vulnerability and Assessment Language
OWASP	The Open Web Application Security Project
OWL	Web Ontology Language
PAL-IS	Palliative Care Information System
PHP	Hypertext Preprocessor
PIM	Platform-Independent-Model
QA	Quality Assurance
RFC	Request for Comments
RUP	Rational Unified Process
SDL	Security Development Lifecycle
SGML	Standard Generalized Markup Language

SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQA	Software Quality Assurance
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URN	Uniform Resource Name
URL	Uniform Resource Locator
US-CERT	The United States Computer Emergency Readiness Team
UTF	Unicode Transformation Format
WASC	Web Application Security Consortium
XML	Extensible Markup Language
XSS	Cross-Site Scripting

Chapter 1. Introduction

1.1. Problem Statement

Over the last two decades, web applications have evolved from web sites with static content, to database-driven and N-Tiered applications that implement complex business logic to provide critical services. Web applications are now the predominant type of enterprise application. Web applications run on the Internet or Intranet, and are open to all web users – both legitimate users and malicious users. There are tremendous amounts of attacks against web applications every day. Often the primary focus for these attacks is to exploit vulnerabilities in the web applications themselves, rather than the operating system or web server where the web applications run (Microsoft Security Development Lifecycle: Risks and Impacts of Computer Crime, 2011).

Vulnerabilities can be introduced into applications at various phases in a traditional software development life cycle due to either flaws in the software development or configuration issues at the time of deployment. Furthermore, as web applications evolve and new technologies are adopted, new types of vulnerabilities, attack vectors and fuzz vectors emerge as the result of either unintended deficiencies in the new technologies or defects in applications due to errors that developers have made when they start to use the new technologies (Allen, Barnum, Ellison, McGraw, & Mead, 2008) (Wysopal, Nelson, Zovi, & Dustin, 2007).

Penetration testing is widely used in industry as a method for security assessment to identify vulnerabilities in web applications. It is often performed by security experts as a post-deployment, isolated test task to measure an application's security posture. It has been proposed by many researchers that penetrating testing should be leveraged for security assurance when an application is still under development, and it is well recognized that the testing should be performed early in a software development life cycle so that any security defects can be fixed with less cost (Arkin, Stender, & McGraw, 2005) (Potter & McGraw, 2004). This is especially true for those vulnerabilities that are introduced by deficient analysis and design (Thomas & Chase, 2005) (Bishop, 2007).

However, in current practice, penetration testing often suffers from the following issues:

1. It is an isolated test process and is often performed late at the application verification phase or in some cases after it is deployed for user acceptance testing. Testers cannot systematically leverage the collaboration with application developers and cannot utilize development artifacts for testing purposes. This makes the penetration testing time-consuming and error-prone.
2. Penetration testing is considered an art. In-depth web security knowledge, solid penetration test skills and extensive experience are essential to a competent tester. Most of time, an internal test team does not have the required web security expertise. While external security experts can be invited to perform a penetration test campaign as a method of security assessment, they are usually not available on a continuous basis during the life cycle of the applications. It is very expensive to keep them in the application project for anything other than short, scheduled assessments.

3. A test campaign is managed as a one-time task. There is no consideration for test maintenance and regression testing. In an iterative development process, tests need to be re-executed many times and regression testing takes significant effort. It is important to take into consideration the maintenance and reuse of existing test artifacts for regression testing.

1.2. Thesis Motivation

To address the issues stated in section 1.1, a penetration test framework for web applications is required that provides guidance to perform systematic and cost-efficient test campaigns, supports testers with reasonable security expertise to produce quality test results, and supports systematic test campaign management. Such a test framework should have the following characteristics:

1. It provides guidance and support to general penetration testers to perform systematic and cost-efficient penetration testing in a security-oriented software development life cycle. Penetration testers can collaborate with developers and utilize development artifacts to improve the efficiency and effectiveness of test campaigns.
2. Web security expertise brought in by external security experts and test artifacts created in a test campaign can be captured and retained in the test campaigns. The retained security expertise can be reused by general testers, who usually are not security experts, to perform test campaigns and produce quality and consistent test results.

3. It supports systematic test campaign management, including test maintenance and regression testing by reusing existing test artifacts.

1.3. Thesis Contributions

In this thesis, we proposed a model-driven penetration test framework for web applications to fill the gap in methodology and tool support for effective and efficient web application penetration test campaigns that are integrated into a security-oriented software development life cycle. General testers with reasonable security expertise can leverage the test framework to efficiently produce quality test results that are reproducible, reliable and assessable. The research in this thesis makes the following contributions:

1. Proposed a systematic web application penetration test methodology that is fully integrated into a security-oriented software development life cycle. The methodology defines three fundamental roles in a test campaign – security expert, developer, and penetration tester; identifies the key development artifacts utilized and the key test artifacts produced in the test campaign; and specifies an iterative test process throughout a security-oriented software development life cycle. The proposed methodology facilitates the collaboration between testers and developers, and the utilization of security, test and development artifacts.
2. Proposed a model-driven penetration test framework that reduces the complexity of test development and increases the reusability of test artifacts in support of the above methodology. The framework leverages a grey-box test

architecture for classic AJAX web applications, a simple web security knowledgebase to capture and retain security expertise, and a penetration test campaign model that captures and maintains the test artifacts throughout the development lifecycle of a web application.

3. Proposed and implemented a prototype Knowledge-based PenTest Workbench, in support of the above framework, that includes a set of tools and utilities in an integrated test environment to automate some aspects of penetration testing, a simple Web Security Knowledgebase, and a Test Campaign Database.

Some aspects of the above contributions have been published in the following papers:

- **Pulei Xiong**, Liam Peyton, "A Model-Driven Penetration Test Framework for Web Applications", PST2010 Eighth Annual Conference on Privacy, Security and Trust, Ottawa, Canada, August, 2010. pp 173-180
- **Pulei Xiong**, Bernard Stepien, Liam Peyton, "A Model-based Penetration Test Framework for Web Applications Using TTCN-3", 4th International MCeTech Conference on eTechnologies, Ottawa, Canada, May, 2009. LNBIP 26, Springer, pp 141-154

1.4. Thesis Methodology and Limitations

The methodology used in this thesis was design-oriented research as described in (Hevner, March, Park, & Ram, 2004). By applying the design-oriented research approach, the research was conducted as a "search process to discover an effective solution to a problem" (Hevner, March, Park, & Ram, 2004). The research work was evaluated using methods such as case study and analysis to demonstrate utility over the other approaches.

The thesis research work followed the five stages of design science research: identification of problem relevance, framework design, framework evaluation, revaluation and improvement of framework, and communication and discussion of research (Bell, Cesare, Iacovelli, Lycett, & Merico, 2007). In particular the process in this thesis research includes the following steps:

1. Investigate the literature and industry best practices for web application penetration testing.
2. Identify and analyze the issues in applying the current web application penetration test practices in the context of a security-oriented software development life cycle.
3. Build an initial framework to address the issues.
4. Conduct a penetration test campaign against WebGoat, a deliberately vulnerable, benchmark web application, by applying the initial framework.
5. Evaluate the initial framework based on test result analysis and comparison to current practice, and identify issues to be improved.
6. Refine the framework to address the identified issues.
7. Conduct the second penetration test campaign against AEMS, a real web application for the hospital adverse event management, which was in the middle of its development process.

8. Evaluate the refined framework based on test result analysis and comparison to current practice, and identify issues to be improved.
9. Refine the framework to address the identified issues.
10. Conduct the third penetration test campaign against PAL-IS, a real web application for palliative pain and symptom management, throughout its entire development process.
11. Evaluate the refined framework based on test result analysis and comparison to current practice.
12. Draw conclusions and identify future work.

The focus of our research has been to demonstrate the possibility of creating a framework that enables general penetration testers to perform a systematic penetration testing integrated in a security-oriented software development life cycle with well-structured interactions with developers and systematic leverage of retained web security knowledge. The framework also enables systematic test campaign management.

It has NOT been a focus of our research to create a tool that will enable testers to be better than experts on their own. The results of this thesis will be limited by the fact that there is still a need for a security expert to set up the knowledgebase for the framework and review the results accomplished by testers.

It is also beyond the scope of this thesis to do large industry-size case studies to establish ROI (Return on Investment). Rather we first wish to demonstrate that our approach is viable and has potential before investing in large scale validation. We have focused on improving the workload distribution between experts and tester/developers, retain expert knowledge when it is provided, and systematically define a penetration test methodology integrated with general secure software development methodology.

It has also not been a focus of our research to build a complete model-driven architecture. We do leverage models in the framework and a formal model-driven architecture might be beneficial in future work, but the engineering of the tool support in our framework was not taken to that level of detail. Rather, we wanted to establish the viability of our approach first before tackling such an initiative.

In our future work section, we do discuss what might be involved in taking the results of our research in these directions.

The thesis is organized as follows:

In chapter 2, the background for the thesis is presented on web application architecture and technologies, web application security, software security engineering and model-driven testing. Methods of knowledge representation are also discussed in the context of web application security and penetration testing.

In chapter 3, the context for the issues to be addressed in this thesis is established by briefly reviewing the current industry best practice, including the Microsoft Security Development Lifecycle, and the Common Criteria Evaluation. Web application penetration testing is discussed, the advantages and weaknesses are analyzed, and potential improvements are identified. Other relevant security assessment approaches that have been done in the field are also analyzed.

In chapter 4, the proposed test framework is presented. First, the problem is clearly defined, and a set of evaluation criteria is identified based on identified issues, the literature review, and a gap analysis of current practices with respect to web application penetration testing. Then, the proposed model-driven penetration test framework is discussed in detail, including a penetration test methodology, a grey-box test architecture, a web security knowledgebase, a penetration test campaign model, and a knowledge-based PenTest workbench.

In chapter 5, the implementation of a framework prototype is described in the context of three different test campaign case studies that were undertaken to evaluate the proposed test framework. The three test campaigns are described in detail, including the steps taken to perform the test campaigns by applying the framework prototype, the analysis of the test results, and the refinement of the framework after the second test campaigns.

In chapter 6, the results of the case studies are summarized and the framework is evaluated against the pre-defined evaluation criteria and is compared to the relevant approaches in current practice.

In chapter 7, the thesis research is concluded and future work is identified.

1.5. Security Terms Used in This Thesis

There is a wide variety of literature on security. Not all terms that we use in this thesis are used consistently in the literature. We provide definitions for some specific terms here, to make it clear about the use of these terms in this thesis.

A *vulnerability* is a defect in software or hardware due to poor design, inappropriate or insecure coding techniques, or configuration mistakes that a malicious individual can exploit (Manzuik, Gold, & Gatford, 2007) (Meier, Mackman, Vasireddy, Dunner, Escamilla, & Murukan, 2003).

Vulnerability assessment is a set of activities that identify security liabilities within a system (network, system software, and applications), and verify that no known security vulnerability is present on the target system (Manzuik, Gold, & Gatford, 2007) (Splaine, 2002).

Penetration testing, also known as ethical hacking, is a process that goes one step further to substantiate the vulnerabilities reported during vulnerability assessment by attempting to recreate the trickery and creativity that a real attacker would use (Manzuik, Gold, & Gatford, 2007) (Splaine, 2002).

A *threat* is a potential event in which a set of circumstances that may lead to compromise – damage, loss, or harm to a system or its users (Hollar & Murphy, 2006) (Meier, Mackman, Vasireddy, Dunner, Escamilla, & Murukan, 2003).

An *attack* is a malicious action to exploit a vulnerability or enact a threat (Meier, Mackman, Vasireddy, Dunner, Escamilla, & Murukan, 2003).

A *countermeasure* is a set of controls put in place to protect vulnerable components from attacks (Hollar & Murphy, 2006).

In addition to the terms above, there are a few terms that are frequently used in the context of web application penetration testing, but no precise and formal definitions for them were found during the literature research. Alternatively, we provide informal descriptions for them so that the content of this thesis can be clearly understood.

- *Entry Point*: any place where a potential hacker can interface within the web application to launch an attack, e.g. user input via browser or an HTTP request in the form of HTTP GET/POST request message.
- *Check Point*: any information that can be used to verify if an attack succeeds or not, e.g. an HTTP response message or an entry in a log file at the server-side.
- *Attack Vector*: a scenario of interaction in which a hacker can launch an attack through an entry point against an application.
- *Attack Surface*: the set of attack vectors for a web application defines its attack surface.
- *Fuzz Vector*: specific values that can be used to parameterize or replace part of an entry point as a malicious payload in an attack.
- *Application Footprint* (*Signature*): system information of an application that might be relevant to an attack vector or fuzz vector.
- *True Negative*: if a test indicates that an attack was not blocked, when it should have been blocked (i.e. the test accurately reports that the vulnerability is not handled).
- *False Negative*: if a test of an attack at an entry point indicates that an attack was not blocked, when in fact the attack was not successful. (i.e. the vulnerability is being handled properly, but the test does not report this).
- *True Positive*: if a test indicates that an attack was blocked and it was (i.e. the test accurately reports that the vulnerability is handled).
- *False Positive*: if a test of an attack at an entry point incorrectly indicates that the attack was blocked, when in fact it was not (i.e. the test result is incorrect, there is a vulnerability that is not being handled).

Chapter 2. Background

2.1. Web Application Architecture and Technologies

N-Tier database-driven web applications have become the predominant architecture for both Internet and enterprise applications. Much of the information processed in web applications is private and highly confidential. Security is therefore an important consideration in the development of web applications (Stuttard & Pinto, 2008). A new generation of Web 2.0 applications is also starting to appear. When new technologies, such as AJAX, Flash, and Web Services, are used to build Web 2.0 applications, they create a new set of security concerns, for example with regard to client-side security and information sources (Shah, 2008). Vulnerabilities may exist on either the server or client side of the web application. Web applications are increasingly vulnerable to malicious attacks on the Web over an increasingly profound attack surface, due to their open and sophisticated architecture and complex business logic.

2.1.1 Web Application Open Architecture

Web applications are built upon a client-server architecture and run on public Internet protocols and technologies. The protocols and technologies used can introduce vulnerabilities due to flaws in the technologies or mistakes that developers make when they apply the technologies during web application development.

N-Tier Web Application Architecture

The classic web application accessed from a client browser consists of three main components: a Web Server, a Web Application that runs on the Web server, and a backend Data Store (Palmer, 2007). The architecture of a typical web application consists of tiers, which usually include a presentation tier, a business tier, and a database tier, that logically separate Presentation, Business Logic, and Data processing aspects of the application. This web application architecture is often referred to as an “N-Tier architecture” (Andreu, 2006). In addition, the J2EE architecture also includes a client tier, which could be a simple HTML page displayed in browser or a full application running at the client-side (Singh, Stearns, Johnson, & Team, 2002).

The N-Tiered architecture that separates functionalities over different tiers is significant from a security perspective. Each tier has its own specific vulnerabilities and interfaces that must be protected. Attackers have full control over the client so usually they launch attacks from there and leverage the interactions between the tiers in sophisticated ways to deliver attacks across the architecture from an entry point in one tier to the actual target in another tier.

AJAX Application

AJAX is an important innovation for Web 2.0 applications (Hoffman & Sullivan, 2008). AJAX is a collection of technologies that allows the client-side code of a web application to make asynchronous, non-blocking requests for fragments of the web page. The fragments can be raw data, for example, JavaScript Object Notation (JSON) or Simple Object Access Protocol (SOAP) that is then transformed into HTML at the client, or HTML fragments that are ready to be inserted directly into the HTML document. In either case, after the server fulfills the request and returns the fragment to the client, the client-side code modifies the web page Document Object Model (DOM) to incorporate the new data.

From an architecture perspective, AJAX applications are evenly-balanced – they do not fit into either traditional thin-client architecture or thick-client architecture. AJAX applications have the inherent security vulnerabilities of both architectures. Furthermore, the AJAX architecture has actually created new potential vulnerabilities by impacting application security in three major ways (Hoffman & Sullivan, 2008): AJAX applications usually have more complex architecture; the internal logic of AJAX applications are exposed to the client-side; and the existence of a server API increases the attack surface.

Web Browser

With new web technologies such as AJAX pushing more responsibility onto the client-side, browser security becomes more important in web application security design and testing (Wells, 2007). Although, in general, a lot of vulnerabilities are still discovered on the server, it is now also common to find vulnerabilities at the client-side. Many of these vulnerabilities are related to DOM-based Cross-Site Scripting (Grossman, Hansen, Petkov, Rager, & Fogie, 2007).

A modern web browser is a complex system. Conceptually, advanced components in a browser include user interface, request generation, response processing, content interpretation, cache, state maintenance, authorization etc. (Shklar & Rosen, 2009). The most important thing for security vulnerabilities is that, due to the complexity of the modern browser, it is very difficult to know what behavior a browser will exhibit for a given user input.

2.1.2 Fundamental Web Technologies

Web applications are built upon a pyramid of technologies including protocols, standards, server-side program and script languages, and client-side browser script languages (Shklar & Rosen, 2009). Among them, three mechanisms are fundamental to the World Wide Web (Raggett, Hors, & Jacobs, 1999):

- A uniform naming scheme for locating resources on the Web, e.g. URL.
- Protocols for access to named resources over the Web, e.g. HTTP.
- Hypertext for easy navigation among resources, e.g. HTML.

Uniform Resource Locator (URL)

Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource on the Internet (Berners-Lee, Fielding, & Masinter, 2005). The generic URI syntax consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment. The URI specification (IETF RFC 3986) does not define a generative grammar for URIs. That task is performed by the individual specifications of each URI scheme such as HTTP(s) and FTP.

A URI can be further classified as a locator (URL), a name (URN), or both. A Uniform Resource Locator (URL) is a subset of the URI that, in addition to identifying a resource, provides a means of locating the resource by describing its primary access mechanism (e.g. its network "location"). An HTTP URL uses the "http" scheme to locate network resources via the HTTP protocol. It takes the form as shown in Figure 1

(Berners-Lee, Masinter, & McCahill, 1994) (Berners-Lee, Fielding, & Masinter, 2005) (Fielding, et al., 1999).

```
   http://<host>:<port>/<path>?<searchpart>#<fragment>

   host = the fully qualified domain name of a network host, or its IP address; 
case-insensitive
   port = the port number to connect to; optional, the default is 80
   path = the location on the server of the file or program specified; case-
sensitive
   searchpart (query string) = name/value pairs data to be passed to web 
applications
   fragment (anchor) = a location on the page
```

Figure 1 HTTP URL

HyperText Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems (Fielding, et al., 1999). It is the core communications protocol used to access the Web and is used by all of today's web applications.

HTTP is a client-server, request/response protocol. A client makes an HTTP request to a server using user agents such as a web browser, spider, or other client-side tool. Then, the server retrieves or creates resources such as HTML files and images and responds to the request. In between the user agent and the server there may be several intermediaries such as a proxy (forwarding agent), a gateway (receiving agent), or a tunnel (relay point). All HTTP transactions follow the same general format (Fielding, et al., 1999): each request message and response message has three parts – the request or response line, a header section, and the entity body; typically, an HTTP client initiates a transaction over a connection with a server; and the server responds with a status line followed by a header and optional body content.

HTTP defines eight methods (also known as “verbs”) that specify the desired action to be performed on the identified resource (Fielding, et al., 1999). The most commonly used methods are GET and POST. The GET method retrieves information (in the form of an entity) identified by the Request-URI from the origin server. The POST method is used to request that the origin server accepts the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI.

HTTP defines five categories of status codes used in the response, that is, Informational (1xx), Successful (2xx), Redirection (3xx), Client Error (4xx), and Server Error (5xx) (Fielding, et al., 1999).

Hypertext Transfer Protocol Secure (HTTPS) is a combination of the HTTP and a cryptographic protocol such as SSL/TLS to protect sensitive HTTP transactions on the Internet (Rescorla, 2000). HTTPS ensures reasonable protection from eavesdroppers and man-in-the-middle attacks, provided that adequate cipher suites are used and that the server certificate is verified and trusted. HTTPS requests and responses function in exactly the same way as the counterparts in HTTP.

HTTP natively provides two forms of authentication scheme: Basic Authentication Scheme and Digest Access Authentication Scheme (Franks, et al., 1999). Basic authentication places credentials in unencrypted form in the header section of an HTTP Get request, so usually basic authentication is used with HTTPS to protect itself from eavesdropping. The authentication is processed by the web server.

Most of the time, web applications use their own authentication module. The most common form of web application authentication is form-based HTML by which user credentials (e.g. user name and password) are placed in an HTTP POST message that is sent to the web application for authentication. Same as the HTTP basic authentication, the form-based HTML authentication has to be used with HTTPS to be protected from eavesdropping. Authentication is a common functionality that is frequently attacked by hackers to either bypass authentication or steal user's credentials.

HTTP is a stateless protocol. There are several ways to manage HTTP state such as URL Rewriting and Hidden Field. IETF RFC 2965 specifies a way of using cookies to create a stateful session with HTTP requests and responses (Kristol & Montulli, 2000). It defines three new headers, Cookie, Cookie2, and Set-Cookie2, which carry state information between participating origin servers and user agents.

A cookie is a small amount of data stored either in a text file on client's file system or in memory in a client browser session (Andreu, 2006), in order to keep persistent state information on the client. Cookie is resent to server with each request. Just like authentication, cookies must be protected from eavesdropping. HTTPS can be used to protect cookies during transmission. Cookie data need to be encrypted to protect it from tampering in memory or on the client's file system (Hoffman & Sullivan, 2008).

Hyper Text Markup Language (HTML)

HTML is the predominant markup language for web pages (Raggett, Hors, & Jacobs, 1999). HTML was defined as an application of the Standard Generalized Markup Language (SGML) – a meta-markup language for defining markup languages (Shklar &

Rosen, 2009). The latest version of HTML specification is 4.01, which was released in December 1999. An HTML 4 document must contain a reference to an HTML version, a header section containing document-wide declarations, and the body of the document. The version declaration names the DTD (Document Type Definition) that should be used to validate the document.

XHTML (eXtensible HyperText Markup Language) is a reformulation of HTML 4 in XML 1.0 (XHTML 1.0 Home Page, 2002), which provides better extensibility and interoperability. HTML 5, the latest version of HTML, is under development (HTML 5 Home Page, 2011), which defines a single language for both HTML syntax and XML syntax, and introduces markup and APIs for web applications.

There are two HTML elements used to generate HTTP requests in web applications: <FORM> and link (hyperlink) (Raggett, Hors, & Jacobs, 1999). An HTML form is a section of a document that is usually used to submit content to a server for processing. A link is a connection from one web resource to another. A link that points to a document is evaluated only upon request, while a link that requests to load images is executed automatically when the page is rendered.

Cascading Style Sheets (CSS) is a mechanism for controlling the style such as fonts, colors, and spacing for HTML rendering (Cascading Style Sheets home page).

DOM

DOM (Document Object Model) is a World Wide Web Consortium (W3C) specification that defines the object model for representing XML and HTML structures

(Grossman, Hansen, Petkov, Rager, & Fogie, 2007). HTML pages are represented and processed internally within a Browser using an HTML DOM. When a page is loaded, the browser parses the page into a DOM object. Client-side scripts such as JavaScript can modify the DOM object in response to user input and internal browser events in a manner that changes the browser display and affects application behavior.

Same Origin Policy

The Same Origin Policy is a crucial element of the browser security model (Hoffman & Sullivan, 2008), which states that scripts in one domain should not be able to access resources or run code in a different domain. The term "Origin" is defined by the triplet of Domain Name, Application Layer Protocol, and TCP Port specified in the URL used to access an HTML document (Hoffman & Sullivan, 2008). Two resources are considered the same origin if and only if all these values are exactly the same. JavaScript implements an important extension to the same origin policy for DOM access in that two sites that share a common top-level domain may opt to communicate despite failing the "same host" check (Hoffman & Sullivan, 2008).

Character Encoding System

Character encoding, also known as character set, is a method (algorithm) for presenting characters in digital form by mapping sequences of code numbers of characters into sequences of octets (Korpela, 2006). American Standard Code for Information Interchange (ASCII) and Unicode Transformation Format (UTF) are two of the most popular character encoding systems.

ASCII has a very small character repertoire so it can only represent English and other western languages (Korpela, 2006). It includes definitions for 128 characters: 94 printable characters including English letters “A” to “Z” and “a” to “z” and digits “0” to “9”, and 33 non-printing control characters. ASCII is a 7-bit character code that each code number can be represented as an integer in binary notation using 7 bits.

Unicode has a much bigger character space. It can represent all the characters of other languages, in addition to the ASCII codes. Unicode encoding model has four levels (Korpela, 2006): an Abstract Character Repertoire that defines the full set of abstract characters; a Coded Character Set that assigns a code number to the character according to a character code; a Character Encoding Form, including UTF-8, UTF-16 and UTF-32, that specifies how the code number is mapped to a sequence of code units; and a Character Encoding Scheme maps code units to strings of octets.

Most modern web browsers can detect character encoding automatically. Hackers often use a variety of character encoding schema, especially together with URL encoding mechanism (discussed below), to construct malicious data for attacks that can bypass web application security mechanisms (such as white list, black list, and sanitization that are discussed in more detail in section 2.2.1) and still can be recognized and accepted by browsers as either input data or output content in web page.

Percent-Encoding in URL and HTTP

There are a set of “reserved’ characters defined in URI that are used to delimit URI components and subcomponents. When reserved characters need to be used for some other purpose in a URI scheme, the characters must be encoded. URI uses a percent-encoding mechanism (also known as URL encoding) to encode the reserved characters (Berners-Lee, Fielding, & Masinter, 2005). A percent-encoded octet is encoded as a character triplet, consisting of the percent character “%” followed by the two hexadecimal digits representing that octet’s numeric value.

Percent-encoding a reserved character involves converting the character to its corresponding byte value in ASCII and then representing that value as a pair of hexadecimal digits. For a non-ASCII character, it is typically converted to its byte sequence in UTF-8. Characters from the unreserved set, or reserved characters that have no reserved purpose in a particular context, may also be percent-encoded but are equivalent to those that are not.

Percent-encoding is also used to encode data of the "application/x-www-form-urlencoded" media type, which is the default content type for HTML form data that is sent to the server in an HTTP request message (Raggett, Hors, & Jacobs, 1999).

2.1.3 Web Application Functionality

With the support of a large number of web technologies, web applications can implement rich functionalities to support complex business logic. The content presented by a web application to users is generated dynamically on the fly, and is often tailored to each specific user. Most N-Tier web applications provide a set of common functionalities to make the business logic available to end users. These include Login and Logout, Session Tracking, User Permissions Enforcement, Role Level Enforcement, Data Access such as HTML form-based data collection and Search, and Application Logic (Palmer, 2007). The implementation of these functionalities can cut across all the different components of a web application architecture including browser, web server, services, and database. They may also be geographically and organizationally distributed in that each component may be located on servers that are geographically separate, and different organizations can be responsible for the processing logic.

Any flaws in any aspect of the implementation of a functionality may cause a security vulnerability. Hackers often try to exploit these vulnerabilities against a web application by analyzing the application's functionalities to identify potential vulnerabilities and then launch attacks to attempt to exploit them.

2.2. Web Application Security

Web application security is a broad topic that encompasses many software development disciplines, technologies, and design concepts (Grossman, Hansen, Petkov, Rager, & Fogie, 2007). In this thesis, we focus on addressing security issues in the testing of customized web applications throughout a security-oriented development lifecycle.

2.2.1 Core Defense Mechanisms

Before the current popularity and proliferation of web applications, efforts to secure organizations against external attacks were mainly focused on the network perimeter. Security measures like firewalls, SSL, intrusion detection systems, network scanners, and passwords were used. However, these technologies are not effective to protect the system against insecure web applications since they are not applied up to the application layer (Grossman, Hansen, Petkov, Rager, & Fogie, 2007). In fact, web applications have become the most popular attack route for system compromise (Hoffman & Sullivan, 2008).

Each web application is different and may contain unique vulnerabilities. Developers and testers have to take responsibility for securing the web applications against vulnerabilities like Cross-Site Scripting (XSS) and Structured Query Language (SQL) Injection that malicious users (hackers) may exploit. During application development, developers need to consider the following questions (Palmer, 2007): if a web application can be compromised due to vulnerabilities within the application; if users of a web application can be compromised due to vulnerabilities within the application; and if the web server or other web services that communicate to a web application be compromised due to vulnerabilities within the application.

The core security problem in web applications is due to the nature of the web application client: the browser (Stuttard & Pinto, 2008). Because the browser is outside of a web application's control, users have full control over the application code at the client-side and can submit completely arbitrary input to the server-side application. The most common Web application security weakness is the failure to properly validate inputs from the client-side (OWASP Testing Guide, 2008). This weakness leads to major vulnerabilities in web applications such as Cross-Site Scripting, SQL Injection, interpreter injection, file system attacks, and buffer overflows. Complex applications often have a large number of entry points, which makes it difficult for developers to validate every possible input.

A number of security mechanisms have been employed to address the fundamental security problem with web applications that all user input is untrusted. These include (Stuttard & Pinto, 2008): restricting user access to an application's data and functionality by applying security measures such as authentication, session management, and access control; validating user input to an application's functions by applying security mechanisms such as *white-list* , *black-list* , and *sanitization*; defending targeted attacks to ensure that an application behaves appropriately by means of appropriate error handling, audit logs, security alters, etc.; and managing an application itself by enabling administrators to monitor its activities and configure its functionality.

The effectiveness of these security mechanisms is dependent on the run-time environment. For example, web applications usually employ different types of input filtering mechanisms to validate users' input. It is important for the input filtering control to be aware of the character encoding (discussed in detail in "Character Encoding System" of section 2.1.2) that is used. If the filter can only detect UTF-8 encoded injections, a different encoding scheme may be employed to bypass the filter.

Classical web application vulnerabilities are still the primary sources of vulnerabilities for AJAX applications. However, the architecture of AJAX introduces new attack possibilities, and it makes many existing web application vulnerabilities more easily exploited due to the increased granularity and transparency of AJAX applications (Hoffman & Sullivan, 2008).

The attack surface of an AJAX application is essentially composed of the complete attack surface of a classical web application and the complete attack surface of web services (extra server-side functionalities) invoked asynchronously by the client (Hoffman & Sullivan, 2008), including: Form Inputs, Hidden Form Inputs, Cookies, Headers, Query Parameters, and Uploaded Files (the attack surface of a classical web application); and Individual parameters of the methods in the client-side scripts, which asynchronously call server-side AJAX functionalities (the attack surface of web services).

2.2.2 Basic Attack Categories and Common Vulnerabilities

Attacks against web applications typically fall into three high-level categories (Hoffman & Sullivan, 2008): resource enumeration, parameter manipulation, and a third category that includes all the attacks that cannot be classified into the two categories above. The third category can be further classified as weakness in session management such as Cross-Site Request Forgeries (CSRF), user credential attack such as phishing scams, Denial of Service (DoS) attacks, etc.

Resource enumeration is used to find content that may be present on the server but is not publicly advertised. There are two types of resource enumeration: one is called blind resource enumeration that makes educated guesses for commonly named files or directories based on pre-defined lists that contain these common files and directories. Web application vulnerability scanners such as Nikto (Nikto Home Page, 2011) maintain and use such lists. A more advanced form of resource enumeration is called knowledge-based resource enumeration. This form of resource enumeration makes more educated guesses based on known web pages or directories on the site, e.g. searching for backup files.

Parameter manipulation is used to manipulate data sent between a browser and a web application to inject malicious code into the server logic or client logic, where the code is then executed or stored, or the code is sent as part of the response back to a victim's browser where the code gets executed. The canonical examples of this type of attack include Cross-Site Scripting and SQL Injection.

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is the most prevalent vulnerability on the Web (OWASP TOP 10, 2007). XSS vulnerabilities exist in web applications that fail to validate, filter, escape, and encode user-controllable input before it is placed in output that is used in a web page (Grossman, Hansen, Petkov, Rager, & Fogie, 2007). XSS violates the intention of the web browser's "same-origin policy".

Cross-Site Scripting is a unique parameter manipulation attack that gets executed in the browser, while in all of the other parameter manipulation attacks the injected code gets executed on the web server (Hoffman & Sullivan, 2008). In XSS, the user is the intended victim, not the server. XSS allows attackers to execute script in the victim's browser to hijack user sessions, deface web sites, insert hostile content, conduct phishing attacks, or take over the user's browser, etc. The malicious script is usually HTML tags and JavaScript, but any scripting language supported by the victim's browser, such as VBScript, Flash and ActiveX, is a potential target for this attack.

There are three main types of XSS: Reflected, Stored, and DOM Injection (Grossman, Hansen, Petkov, Rager, & Fogie, 2007) (OWASP Cross-site Scripting, 2011) (OWASP DOM Based XSS, 2011). Reflected XSS attacks, also known as non-persistent

XSS attacks, are those where the injected code is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web server. Stored XSS attacks, also known as persistent XSS attacks, are those where the injected code is permanently stored on the target servers, e.g. in a database. Stored XSS attacks most often occur in either community content-driven web sites or web mail sites. DOM-based XSS are those where the attack payload is executed as a result of modifying the DOM in the victim's browser by the original client-side script. AJAX applications push much of the business logic to the client. Therefore, the chances of finding DOM-based XSS in AJAX applications are quite high. DOM-based XSS vulnerabilities can be persistent or non-persistent. DOM-based XSS is unique form of XSS in that the malware payload does not get sent to web server, therefore the general XSS filtering and detection mechanisms are impotent against such attacks.

XSS vulnerabilities occur when different encodings are used. Various input filters employed in web applications can be evaded or bypassed by encoding the input into something that is understandable by the browser and completely valid for the filter. In addition, browsers differ in how they render HTML and JavaScript. It is very common that one vector will work in one browser, yet not work in another.

To detect XSS vulnerabilities, a tester will analyze each input vector and use specially crafted input data with each input vector. Such input data is typically harmless, but triggers responses from the browser that manifest the vulnerability. A complete test will include instantiating a variable with several attack vectors by applying various fuzz vectors. Automated testing such as web spiders and vulnerability scanners can detect only the simplest XSS vulnerabilities. Persistent and DOM-based XSS vulnerabilities are almost always missed (Grossman, Hansen, Petkov, Rager, & Fogie, 2007)(OWASP Testing Guide, 2008).

SQL Injection

SQL Injection is a type of injection attack where SQL commands are inserted via the input data from users to a web application in order to affect the execution of predefined SQL commands (OWASP Testing Guide, 2008). SQL Injection allows the execution of SQL code under the privileges of the user used to connect to the database. A successful SQL Injection exploit can read sensitive data from the database, modify database data, execute administration operations on the database, recover the content of a given file existing on the DBMS file system and, in some cases, issue commands to the operating system.

The use of dynamic SQL opens the door to SQL Injection attacks (OWASP Testing Guide, 2008). If user input is not filtered for escape characters, malicious code can be passed into a SQL statement. The techniques used to attack a specific database vary by the brand of the DBMS and the version of the DBMS since different databases use some different functions, syntax, and comment styles.

There are three classes of SQL Injection attack techniques to retrieve data related to a web application (OWASP Testing Guide, 2008): the most straightforward type of attack called Inband is to retrieve data that is presented directly in application web pages;

a way called Out-of-band is to retrieve data using a different channel, e.g. an email; and a way called Inferential is to reconstruct information based on the data collected by sending particular requests to a web application or DB server and observing the resulting behavior.

SQL Injection attacks that manipulate the existing query criteria and/or parameters are classified as “Standard SQL Injection”. Attacks that involve the use of the UNION operator to join a purposely forged query to the original query are called “Union Query SQL Injection” (OWASP Testing Guide, 2008). Another type of attacks is known as “Blind SQL Injection” in that it needs to analyze the logic of the original queries in order to construct an attack.

To detect a SQL Injection vulnerability in a web application, the first step is to understand when an application connects to database. Some application functionalities from section 2.1.3 that are likely to invoke a database connection include Login and Logout, and Data Access such as HTML form-based data collection and Search. Places where a user enters free text into a field that will be used in a SQL query are vulnerable; for example a username field during Login or a search term in Search. A simple test for SQL Injection consists of adding a single quote (') or a semicolon (;) to the field under test and observing the response from the web application (OWASP Testing Guide, 2008).

2.3. Software Security Engineering

Weaknesses in software, such as defects and bad configurations, may cause the software to be vulnerable to malicious attacks. It is generally considered good software engineering practice to adopt a security-oriented software development process to eliminate potential vulnerabilities as early as possible in the software development life cycle (McGraw, 2006) (Howard & Lipner, 2006). As a result, the number of exploitable weaknesses in the software can be reduced before it is released.

2.3.1 Software Security Development Life Cycle

Software security is fundamentally a software engineering issue that must be addressed in a systematic way throughout the software development life cycle (McGraw, 2006), which sometimes is an iterative process e.g. Rational Unified Process (Arlow & Neustadt, 2005).

Defects and other weaknesses in software that are caused by the increased complexity of software systems (including unintended interactions between software components especially those provided by third parties), flawed specifications, flawed designs, or defective implementations, can affect software security (Allen, Barnum, Ellison, McGraw, & Mead, 2008) (Wysopal, Nelson, Zovi, & Dustin, 2007). Adopting a software development life cycle that includes secure development practices can reduce the number of exploitable defects and weaknesses in the software. Through the adoption of a security-oriented software development life cycle, potential vulnerabilities can get addressed more quickly and at a lower cost (OWASP Testing Guide, 2008). This is far more cost-effective than attempting to diagnose and correct such security defects after the system is deployed into production.

2.3.2 Software Quality Assurance

Software Quality Assurance (SQA) is a systematic, planned set of actions to provide adequate confidence that a software system product conforms to established functional and technical requirements, including security requirements, within managerial scheduling and budgetary requirements (Galin, 2004). Cooperation and collaboration between a software development team and other teams such as a QA team is an important consideration in the development of SQA methodologies and their implementation.

Software testing is the primary software quality assurance process applied to control the software product's quality before its release in that it consumes the most software quality assurance resources (Galin, 2004). Tests must be repeatable and reusable. Testing should be integrated into the software life cycle and should be performed efficiently and effectively (Burnstein, 2003). Basically, there are three types of test approaches for test design that aim to achieve different coverage criteria, including black-box, white-box and grey-box.

2.3.3 Software Security Testing

Security testing plays a critical role in a security-oriented software development life cycle (Allen, Barnum, Ellison, McGraw, & Mead, 2008). It ensures that the software being tested is robust and continues to function even in the presence of malicious attacks. Security testing is different from traditional functional testing as there is no system specifications for security testing that can be used directly as test requirements. A few security test methods, including code reviews, risk analysis, automated tools and penetration testing, are widely used in security testing (McGraw, 2006).

Penetration testing is often conducted as a black-box test method that is performed to detect vulnerabilities in information systems (Howard & Lipner, 2006). In industry, penetration testing is typically performed by security experts as a one-time security audit just before or just after it is released from development. However, a few researchers have proposed the idea of integrating penetration testing into the software development life cycle. In (Arkin, Stender, & McGraw, 2005) and (Potter & McGraw, 2004), the authors advocate integrating application penetration testing into a complete security-oriented software development life cycle. In (Thompson, 2005), the author presents a comprehensive penetration testing process, including test analysis (with threat modeling), test design, test execution and report. In (Bishop, 2007), the author discusses the analysis of threats and potential attackers as a means of providing valuable input to perform effective and efficient penetration testing. Neither researcher discussed their penetration testing approach in a level of detail such that it is ready-to-use in industry, e.g. they did not define the methodology used in the integrated penetration test campaign or the design of tools supporting the methodology.

2.4. Model-Driven Software Testing

In model-driven software testing, models are used to express application requirements that can be used to generate test cases. Model-driven test approaches can be used to reduce the complexity in test design and increase test reusability.

2.4.1 Software Testing Fundamentals

"Testing is the process of executing a program with the intent of finding errors" (Myers, 2004). Testing is an expensive task in software development. Risk analysis can be used to choose a cost-efficient test approach to identify the greatest number of critical defects with the least cost (Allen, Barnum, Ellison, McGraw, & Mead, 2008).

There are three basic test approaches: black-box, white-box, and grey-box (Binder, 1999) (Patton, 2000). Black-box testing is a test method in which a tester tests an application based on the specified or expected responsibilities of a unit, subsystem, or system without knowing the details of how the application works. In white-box testing, a tester has access to the application's source code and develops test cases based on the analysis of the source code. Grey-box is a test approach that is a combination of black-box and white-box. An application is still tested as a black-box but supplemented with the application's internal information. It is particularly suited to the testing of modern web applications (Patton, 2000), which are often comprised of many components across an N-Tier architecture. In grey-box testing of web applications, each component of a web application is treated as a black-box but the interactions and communications between components are known and tests can be based on an analysis of those communications.

Testing usually means dynamic testing – running and using an application (Patton, 2000). An application can be tested without running the application. This is called static testing (Patton, 2000). Code review is a static white-box testing technique.

An oracle is a generation and comparison mechanism in software testing which produces the expected results for an input and checks the actual results against the expected results (Binder, 1999). In penetration testing, this means how to identify expected results in a web application for every test attack, and how to verify actual results against the expected results.

Regression testing is a specific test process where selective retesting of an application is performed to verify that no new defects were introduced by any modifications in the application (Binder, 1999) (Patton, 2000). In a security-oriented software development life cycle, penetration testing may need to be re-executed during regression testing if there are any modifications in a web application, or if there are new emerging threats that have to be addressed (OWASP Testing Guide, 2008).

2.4.2 Model-Based Testing

Model-Based Testing is a software testing method that automatically generates test cases and provides an oracle to determine test results from the models that describe application expected behaviors (Jacky, Veanes, Campbell, & Schulte, 2008). Test models can be extracted from application specifications, design artifacts, or program source code (Pezz & Young, 2008). Test case specifications that are generated from the test model are used to reveal discrepancies between actual program behavior and the model. Model-Based Testing achieves systematic and automated software testing (Binder, 1999).

2.4.3 Model Driven Engineering

Model Driven Engineering (MDE) is a systematic approach that can be leveraged to reduce the complexity of application development by utilizing explicit models of the application domain (Schmidt, 2006). MDE technologies usually involve domain-specific modeling languages, which model the key concepts in the domain and the relationships between them, combined with transformation engines or generators that can generate some artifact(s) related to the development of the system such as source code, tests, deployment descriptions etc.

Model Driven Architecture (MDA) is an important research initiative in Model Driven Engineering (Model Driven Architecture Home Page, 2011). In MDA, modeling languages are used as a sort of programming language (Frankel, 2003) that raises the level of abstraction and reuse in software development. A system is first modeled as a Platform-Independent-Model (PIM) using a domain specific language. The PIM is then transformed, usually with the support of automation tools, to one or more Platform-Specific-Models (PSMs), and then transformed into code which is executed in computers (Mellor, Scott, Uhl, & Weise, 2004) (Kleppe, Warmer, & Bast, 2003) (Stahl & Voelter, 2006).

2.4.4 Model-Driven Testing

While Model-Driven Architecture is primarily used for software development, it can be applied to software testing as well to improve efficiency and effectiveness. For example, in (Baker, Dai, Grabowski, Haugen, Schieferdecker, & Williams, 2008), the authors discussed how to apply Model-Driven approaches to functional testing and performance testing at the unit, integration and system levels, as well as for SOA (Service Oriented Architecture) applications testing.

Model-Driven Testing includes test artifacts modeling and automated test cases generation using the typical MDA transformation mechanism (Baker, Dai, Grabowski, Haugen, Schieferdecker, & Williams, 2008) (Dai, 2004) (Javed, Strooper, & Watson, 2007). Model-Driven testing is often combined with data-driven approaches (Baker, Dai, Grabowski, Haugen, Schieferdecker, & Williams, 2008). To the best of our knowledge there is no application of a Model-Driven approach in the field of security testing.

2.5. Web Security Knowledgebase

Vulnerabilities found in web applications in the past can be learned and catalogued for reuse to ensure that the same errors are avoided (Wells, 2007). Web application security knowledge exists in a variety of publications as well as on the Web. However, the format and terminology are not often used in a consistent way. The information is continuously being extended and updated as new vulnerabilities appear. Developers and testers have to keep their web application security knowledge updated so that they are capable of recognizing and uncovering new vulnerabilities in web applications. Ideally, web application security knowledge should be represented in a well-structured format so that it can be maintained and leveraged efficiently and effectively by security experts, developers and testers.

2.5.1 Web Application Security Knowledge Sources

Web application security knowledge, including architecture security, design security, coding security, and security testing, can be found in textbooks, papers, articles etc. The most up to date knowledge is usually available on the Web. Specifically, for penetration testing, there are many publicly available knowledgebases on the Web including OWASP Top 10 Most Critical Web Application Security Vulnerabilities (OWASP TOP 10, 2007), WASC Threat Classification version 2.0 (WASC, 2010), SANS Top 20 Cyber Security Risk (SANS TOP 20, 2011), SNAS Top 25 Most Dangerous Software Errors (SANS TOP 25, 2009), MITRE Common Weakness Enumeration (MITRE CWE Home Page, 2011), MITRE Common Attack Pattern Enumeration and Classification (MITRE CAPEC, 2011). There are also security testing guides such as OWASP Testing Guide (OWASP Testing Guide, 2008), and a large volume of articles and blogs that are posted on the Web by security experts, e.g. XSS (Cross Site Scripting) cheat sheet (Hansen, 2011), SQL Injection cheat sheet (Daw, SQL Injection Cheat Sheet, 2009), and Input Validation cheat sheet (Daw, Input Validation Cheat Sheet, 2009).

These knowledgebases and documents classify known, generic web application vulnerabilities, attack vectors and fuzz vectors. They are fundamental references used by web application developers and penetration testers for self-education and for secure web application development and testing. These knowledgebases exist as text-based, unstructured documents.

2.5.2 Security Knowledge Representation

Web security knowledge is profound and complicated. In general, there are a number of different types of human knowledge, including procedural knowledge, declarative knowledge, structural knowledge, and ontological knowledge (Gaševic, Djuric, & Devedzic, 2006). Ontological knowledge provides a number of useful features for knowledge representation to support knowledge sharing and reuse. It identifies classes of objects, their relations, and concept hierarchies. Most web security knowledge is expressed in natural language. However, the Open Vulnerability and Assessment Language (OVAL) (OVAL, 2011) and Application Vulnerability Description Language (AVDL) (Bialkowski & Heineiman, 2004) are two examples of XML-based languages to represent security knowledge. In this thesis, we build an object model to represent a simple web security knowledgebase, and implement it in a relational database.

Chapter 3. Related Research and Industrial Practice

3.1. Microsoft Security Development Lifecycle

The Microsoft Security Development Lifecycle (the MS SDL) is a software security assurance process that defines a comprehensive security-oriented software development life cycle from a developer's perspective. The MS SDL was developed based on the traditional Microsoft product development process by adding steps to the development process to address all elements of SD3+C – the Microsoft secure development principles: Secure by Design, Secure by Default, Secure in Deployment, and Communications. The MS SDL consists of the phases of training, requirements, design, implementation, verification, release, and response (see Figure 2 excerpted from Microsoft Security Development Lifecycle Version 5.0). It also defines a set of mandatory security activities with tool support (many are based on Microsoft .Net platform) to leverage a variety of security test methods, such as threat modeling, static code analysis, and dynamic fuzz testing (About the SDL Process, 2011) (Howard & Lipner, 2006).

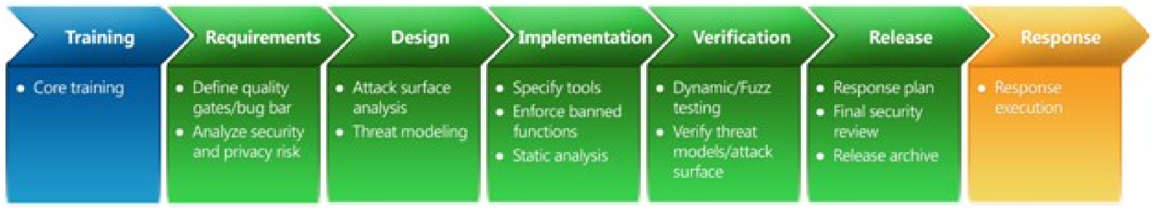


Figure 2 Secure software development process model at Microsoft

In the MS SDL, penetration testing is an optional task at the verification phase that is performed independently (isolated from the MS SDL) by security experts external to the development team (Simplified Implementation of the Microsoft SDL, 2010) (Microsoft Security Development Lifecycle Version 5.0, 2010). Any issues identified by penetration testing must be addressed and resolved before an application can be released. Threat modeling can be used to prioritize tests (attacks) against an application (Microsoft Security Development Lifecycle Version 5.0, 2010). The MS SDL though does not define "how" the penetration testing should be performed and does not suggest any tools to be leveraged. It relies on the external security expert resource to apply the best industry practice. It is not supposed to retain the external expert's knowledge and test artifacts to be reused by internal test team later on when re-running tests is required.

3.2. The Common Criteria Evaluation

The Common Criteria (CC) is an international standard ISO/IEC 15408 (ISO Freely Available Standards, 2011) that defines a set of commonly agreed criteria for security evaluation and certification of IT products or systems. The CC originated out of three standards: the European standard ITSEC, the Canadian standard CTCPEC, and the United States standard TCSEC (The Common Criteria Introduction). The US TCSEC, also known as the Orange Book, was used to evaluate and classify computer systems for processing sensitive or classified information (The US DoD Trusted Computer System Evaluation Criteria, 1985). The Orange Book was cancelled in 2002 (Department of Defense Directive Number 8500.01E, 2002).

The CC is currently in version 3.1. The CC standard includes three parts: Part I Introduction and general model (The Common Criteria Part I: Introduction and general model, 2009), Part II Security functional requirements (The Common Criteria Part II:

Security functional requirements, 2009), and Part III Security assurance requirements (The Common Criteria Part III: Security assurance requirements, 2009). In the framework of CC evaluation, vendors implement and make claims about the security features and attributes of their products specified based on the standard functional requirements defined in the CC part II; security evaluation laboratories evaluate the products based on the CC part III assurance requirements to gain confidence at a certain Evaluation Assurance Level (EAL), from the entry level 1 to the top level 7, that the claimed security measures are effective and have been implemented correctly; customers can select desired products based on the claimed security functions and the evaluation assurance level (The Common Criteria Part I: Introduction and general model, 2009).

A common methodology (CEM) (The Common Methodology for Information Technology Security Evaluation, 2009) was developed, as an effort of international collaboration, for guiding the IT security evaluation based on the CC standards. The CEM defines an evaluation methodology for only up to EAL4, the highest level that is recognized by all the members of the Common Criteria Recognition Arrangement (The Common Criteria Portal Home Page, 2011). Accredited security laboratories independently conduct CC evaluation by following the CEM. The evaluation (up to EAL4) is an isolated process from the product development. The evaluation activities include document review, very limited code review, site visit, and independent functional testing and vulnerability assessment (penetration testing). The CEM does not touch upon any tool support or security knowledgebase for the penetration testing (as well as all other evaluation activities), and it does not have considerations for test re-execution, e.g. for the assurance continuity.

The CEM is supposed to be used by certified CC evaluators, rather than general testers. CC evaluation is supervised under a schema that is developed by each member of the Common Criteria Recognition Arrangement. In Canada, it is Common Criteria Schema (CCS) developed and maintained by Communications Security Establishment Canada (Common Criteria Schema Overview). The CCS ensures CC evaluation is performed to a high degree of quality. Specifically, it relies on the expertise and integrity of certified evaluators to achieve such a high quality: a certified evaluator should possess a suitable combination of IT security education and relevant experience and receive a passing grade on a CCS evaluator exam, and has been granted security clearance (Common Criteria Scheme Guide #3 Evaluation Facility Approval, 2010).

3.3. Web Application Penetration Testing

Penetration testing is widely used in industry to test web application security. It can be used to identify "known" vulnerabilities in off the shelf applications that exist in a network, or it can be used to discover "unknown" vulnerabilities in custom-built web applications. Our thesis is primarily concerned with the latter, but it is still relevant to understand both types.

When the purpose of penetration testing is to identify any "known" vulnerability in off the shelf applications running in a network, it falls into the category of network security assessment. Open source network scanners such as Nmap (Nmap Home Page, 2011) and vulnerability scanners such as Nessus (Nessus Home Page, 2011) can be used together to reconnaissance a network, enumerate the live services (including web applications) on the network, and identify any known vulnerabilities logged in open source vulnerability databases such as OSVDB (The OSVDB Hoem Page, 2011), US-CERT (US-Cert Vulnerability Notes Database Home Page, 2011), NVD (NVD Home Page, 2011), Bugtraq mailing list (Bugtraq Home Page, 2011), and MITRE CVE (MITRE CVE Home Page, 2011). A number of methodologies have been created which guide a tester through the process of doing such testing, of which a typical methodology is the ISECOM Open Source Security Testing Methodology Manual (ISECOM OSSTMM Home Page, 2011).

Penetration testing to discover vulnerabilities that have never been identified in a custom-built web application is often performed by security experts as a method of security assessment for the application. There are also tools available that experts will use when performing penetration testing, although increasingly they are being automated and promoted as stand-alone tools which can be used without the help of an expert – a set of such tools is discussed in detail in section 3.3.3.

In the next a few subsections, we give an overview of several penetration methodologies being used in industry, and summarize these industry best practices as Expert Security Assessment (ESA) in section 3.3.2 that is used as a representative composite of the current industrial best practices.

3.3.1 Best Practices in Industry

In this section, we present several best practices in industry based on our literature research, including three well-sold textbooks on penetration testing and an open source security testing guide from OWASP.

Web Application Vulnerabilities Detect, Exploit, Prevent

In this book (Palmer, 2007), the authors present a penetration test (hacking) methodology that includes:

- intelligently scanning the application server for relevant information of deployment and configuration
- walking through the application to identify all of the interactive pages and related functionalities, parameters, GET/POST methods etc. that the pages take
- fuzzing the identified parameters – manipulate it to be used for attacks
- running tests to validate vulnerabilities
- reporting every vulnerability and security related issue that was found

The authors also describe in detail some categories of common web application vulnerabilities and sample attack vectors, including server-side vulnerabilities such as XSS and client-side scripting vulnerabilities such as those related to JavaScript and CGI.

The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws

This book (Stuttard & Pinto, 2008) explicitly defined a web application penetration test methodology:

- during the *Recon and Analysis phase*, a tester explores and analyze a web application to identify its functionalities, entry points, and technologies being used such as client side scripting, backend HTTP server and database etc. (footprint), and then identify potential vulnerabilities based on the functionalities to further construct attack surface
- during the *Test Execution phase*, for each identified potential vulnerabilities, launch attacks based on the categorized attack vectors

The attack vectors are categorized aligned with Application Logic including client-side controls and logic flaws, Access Handling including authentication, session management and access control, Input Handling, Application Hosting such as shared hosting issue and web server security issue, and Miscellaneous security issues.

Professional Pen Testing for Web Applications

The author defined an informal penetration test methodology in this book (Andreu, 2006). It includes the following steps:

- discover and analyse system information (OS footprint, web server footprint etc.) to understand the test target
- discover application footprint (technology/platform), enumerate entry points, and discover areas of weakness
- launch attacks on target to probe for areas of weakness
- analyse attack results to verify discovered vulnerabilities
- document test results and present them with recommendations to the project stakeholders

OWASP Testing Guide

OWASP Testing Guide version 3.0 (OWASP Testing Guide, 2008) was developed by a group of security experts to provide a best practice of web application penetration testing with low-level guidance. It was released as an open source document.

The testing is divided into major two phases (OWASP Testing Guide, 2008): Passive Mode and Active Mode. The passive mode is for a tester to understand an application's logic and play with the application for information gathering such as testing for application’s footprint and identifying entry points. The active mode is to run tests based on well-categorized attack vectors by applying appropriate fuzz vectors.

At the end of the tests, the tester will estimate the risks to the business of any identified vulnerabilities based on factors such as likelihood and impact, and then create a well-informed test report.

3.3.2 Expert Security Assessment

The test methodologies defined/described in the literatures (Stuttard & Pinto, 2008) (Palmer, 2007) (Andreu, 2006) (OWASP Testing Guide, 2008) may look slightly different from each other, e.g. vary on the name of the test phases and test activities, or the activities divided in each phase, but they essentially talk about the same things. That is, at a high level, a security expert typically will perform Information Gathering to understand the characteristics (footprint) of an application and its running environment, Application Analysis to enumerate entry points and identify their functionalities, Test Construction and Execution to create concrete test cases by applying feasible attack vectors and fuzz vectors, and Test Reporting to document discovered vulnerabilities. In practice, the exact set of steps will vary between experts and between the specific applications they are testing. Below we describe in detail a prototypical penetration methodology that a security expert will go through.

At a first step, the expert attempts to gather information about the application and its running environment including:

- Examining client-side content, such as HTML and script.

- Installing a web application proxy, e.g. WebScarab (OWASP WebScarab, 2011), and then configuring the browser to access the server via the proxy.
- Analyzing communications between the client and server, e.g. how a user is authenticated and how cookies are used to manage application state.
- Sending malformed requests to the server and attempting to receive verbose error messages, either from prompting error message windows or from the server's responses, which reveal the application footprint and information about components in the application architecture.

Second step, the expert walks through the application to enumerate all entry points and map them to web application functionalities (as described in section 2.1.3):

- Using a web application proxy to monitor the HTTP traffic between the browser and the application, the expert documents all the HTTP requests sent to the server and the responses received.
- Mapping each HTTP request to a web application functionality.

Third Step, the expert launches penetration tests against the web application:

- Based on the functionality tagged to each HTTP request, the expert identifies any potential vulnerabilities of which the expert has knowledge.
- For each potential vulnerability, the expert lists all attack vectors that they are aware of.
- For each attack vector, the expert uses all fuzz vectors that they are aware of to create specific tests.
- The expert has a few ways to run the tests:
    - launch the tests through the browser directly
    - use browser extensions, e.g. FireBug, to intercept and manipulate DOM events or client-side scripts, and then send them out
    - use a web application proxy to intercept and manipulate outgoing HTTP requests, and then send them out

    - develop and use a customized client application
- The expert then observes the responses at the client-side by either looking at the HTML content rendered in the browser, or inspecting DOM in memory, or checking the raw HTTP responses captured in the web application proxy. Based on what is observed, the expert determines if the attacks succeeded or not to confirm the existence of the vulnerabilities.
- The expert may use various tools to assist his tests during the testing, e.g. using an XSS cheat sheet or a SQL Injection cheat sheet to pick up attack vectors and fuzz vectors, and using an Encoder, e.g. the built-in encoder called Transcoder in WebScarab, to help manipulate parameters in HTTP requests.

Finally, the expert collects and analyzes the vulnerabilities that have been found, and creates a report.

These activities are challenging to general testers who usually do not have adequate security expertise to perform these steps on their own. The coverage of the test cases and the quality of the tests are determined by, or constrained by, the testers' knowledge, skills and experience in web application security (Arkin, Stender, & McGraw, 2005). In addition, real hackers have more resources than general penetration testers (Bishop, About Penetration Testing, 2007). Taken as a whole, hackers all over the world have more comprehensive and in-depth knowledge, more solid skills and more extensive experience in web application security than the average team of general purposes testers. Hackers usually are not restricted to any tight schedule, and they only need to detect and then exploit a single vulnerability in a web application in order to compromise the application, while the test team in an application development project is expected to uncover all vulnerabilities in the application within an aggressive project schedule.

3.3.3 Tool Support

There are lots of tools or utilities, either open source or commercial, that penetration testers can leverage to help their test efforts. Many books on penetration testing, for example “Professional Pen Testing for Web Applications” (Andreu, 2006), recommend a toolkit that has a variety of different types of utilities for different aspects of penetration testing, e.g. web spiders (crawlers) that traverse a web application automatically to collect possible HTTP requests to the web application, HTTP proxies that act as an interacting service between browser and web server to capture and intercept HTTP requests and responses, fuzzers that generate invalid, unexpected, or random data to be used for input parameter manipulation.

There are also utilities designed for addressing specific vulnerabilities. For example, several Firefox extensions including DOM Inspector, Web Developer, FireBug, LiveHTTPHeaders, ModifyHeaders, and TamperData, can be used for XSS detection and exploitation (Grossman, Hansen, Petkov, Rager, & Fogie, 2007); SqlDumper is recommended to generate high volume of queries for blind SQL Injection attacks (OWASP Testing Guide, 2008). There are several client-side web application exploitation frameworks that integrate different techniques to detect, exploit, and provide insight into the problems in web applications, e.g. AttackAPI, BeEF, CAL9000, and XSS-Proxy (Grossman, Hansen, Petkov, Rager, & Fogie, 2007).

These tools and utilities are from different vendors, they are not designed based on a common, integrated platform. When they are used for different aspects of penetration testing, testers have to manually interpret and convert the output of one tool to the input of another tool that is used at a following step.

3.4. Other Security Assurance Approaches

In addition to penetration testing, there are a variety of other approaches for web application security testing, including (OWASP Testing Guide, 2008)

- manual, static inspection approaches such as **Code Reviews** that systematically examines source code of a web application to improve the overall quality of the application and the developers' skills;
- automated, static approaches such as using a **Code Analyzer** that scans source code of a web application to discover weakness e.g. unsafe API in the code;
- and automated, dynamic approaches such as using a **Vulnerability Scanner** that scans a running web application for potential vulnerabilities.

Manual inspection approaches, such as code reviews, can be effective if they are done systematically and thoroughly by security experts (OWASP Testing Guide, 2008). In some cases, code reviews can be more efficient if they are used with penetration testing (OWASP TOP 10, 2007). However, in the cases where source code is not accessible, code reviews are not feasible. And in the cases where the code base is large, code reviews can be time-consuming and error-prone.

Automated tools, including code analyzers and vulnerability scanners, are very efficient for some specific vulnerabilities, e.g. a vulnerability scanner is efficient to detect "resource enumeration" attacks (Hoffman & Sullivan, 2008). Usually, vulnerability scanners have their own propriety vulnerability knowledgebase (Palmer, 2007). Security experts may leverage vulnerability scanners during their security assessment.

While the automated tools can be quite efficient on certain security issues, the customized nature of most web applications significantly reduces the effectiveness of the tools. These tools can't detect all types of vulnerabilities, especially those deeply intertwined in business logic and custom application design, e.g. cross-site scripting (OWASP Testing Guide, 2008) (Grossman, Hansen, Petkov, Rager, & Fogie, 2007). And these tools are not necessarily faster than manual methods because it takes much time to set up the tools before running and it takes a significant amount of time after running to analyze a large number of reported issues that turn out to be non-issues (OWASP Testing Guide, 2008) (Scambray, Shema, & Sima, 2006). So far, no automation tool can replace the skills of an experienced security expert, and the most effective tools should be chosen carefully for their performance on specific types of vulnerabilities (Sutton, Greene, & Amini, 2007).

Chapter 4. Model-Driven Penetration Test Framework

4.1. Problem Definition

Security is a critical aspect that needs to be considered at the very beginning when a web application is developed. Currently, such penetration testing effort is typically undertaken by external security experts as an isolated testing campaign performed at a late phase of a software development life cycle. For example, in the MS SDL, penetration testing is an optional task that is performed by external security experts as late at the verification phase that is just before the release phase (as presented in detail in section 3.1). Many researchers have proposed to leverage penetration testing as a test method for web application security assurance when the application is still under development. However, there is no well-defined test methodology which guides penetration testers on how to perform a systematic penetration test campaign within the context of a security-oriented software development lifecycle.

Security experts typically run a penetration test campaign against a deployed web application by following the steps of information gathering, entry point enumeration, test execution, and test result analysis and reporting (as presented in detail in section 3.1). The security experts have in-depth knowledge, solid skills and extensive experience in web application security. To uncover as many vulnerabilities in the application as they can, the security experts work diligently and creatively with out-of-the-box thinking and apply their vast security expertise appropriately to develop test cases, execute tests, and interpret test results. Specifically, when the security experts develop test cases, as shown in Figure 3, for each *Entry Point* they need to determine:

- All potential *Vulnerability* based on the *Application Footprint* and possible *Attack Vectors*. (See section 1.5 for definitions of italicized words)
- For each attack vector, which *Fuzz Vectors* can be applied based on the *Application Footprint* to instantiate *Test Cases*.
- Finally when each *Test Case* is executed, which *Check Points* can be used to determine *Test Results* (whether a *Vulnerability* is confirmed or not).

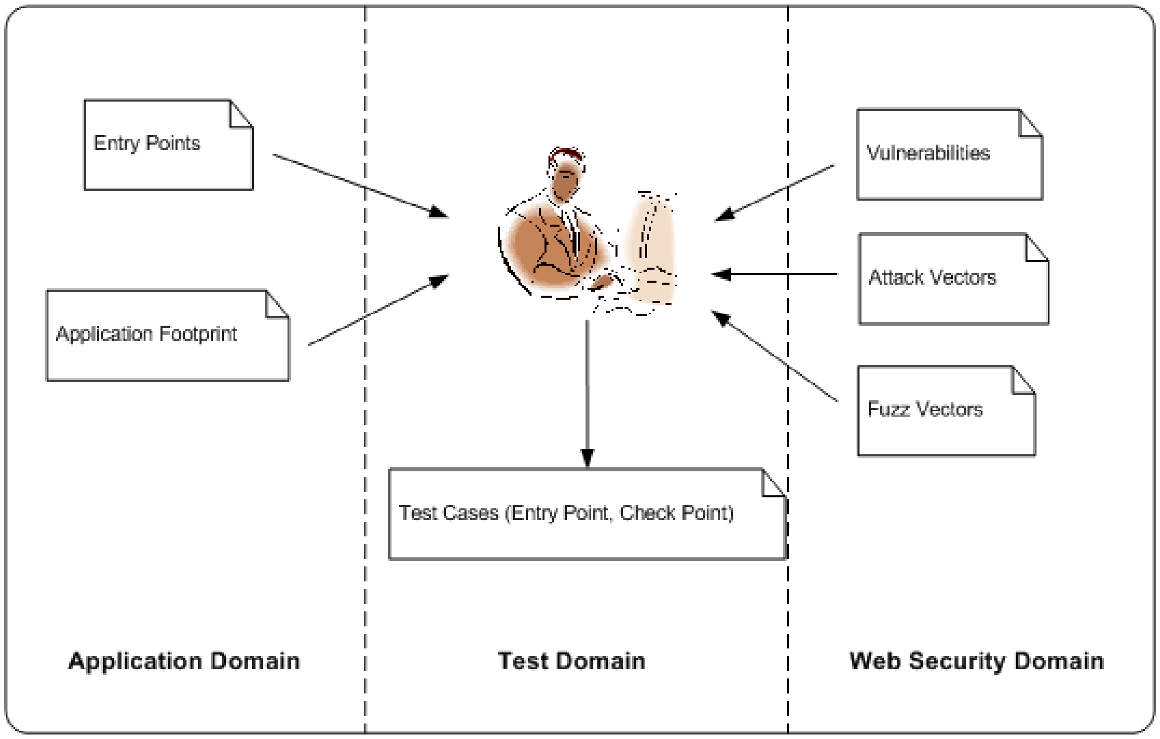


Figure 3 Penetration Test Development – Current Practice

Based on this approach, the quality of a test campaign relies on the expertise and diligence of the security experts. Considering an application development project may take months or even longer, external security experts may not always be available, and even if they are available, it is very expensive to keep them in the project all the time. However, on the other hand, general testers in an internal test team usually do not have the required security expertise to successfully accomplish such a penetration test campaign.

In addition, in such a penetration test campaign, the testing performed by security experts is often a one-time, post-deployment task. There is no need to re-run the tests against the same application many times. But in a software development life cycle, a web application is developed in an iterative process. Test cases need to be updated from time to time in various situations, such as when new features are added in, implementation is changed due to bug fixes, or new vulnerabilities, attack vectors, and fuzz vectors are discovered. The test cases therefore need to be updated and re-run frequently to verify that bugs have been fixed appropriately and there is no regression in the application that have introduced new bugs. Test maintenance and regression testing both need significant effort and support.

In this thesis, we propose a model-driven penetration test framework for web applications that focuses on addressing the issues discussed above:

- Defines a test methodology that guides and supports general penetration testers to perform systematic and cost-efficient penetration testing in a security-oriented software development life cycle.
- Captures and retains web security knowledge and test artifacts so that they can be reused by general testers to produce quality and consistent test results.
- Supports systematic test campaign management, including test maintenance and regression testing by reusing existing test artifacts.

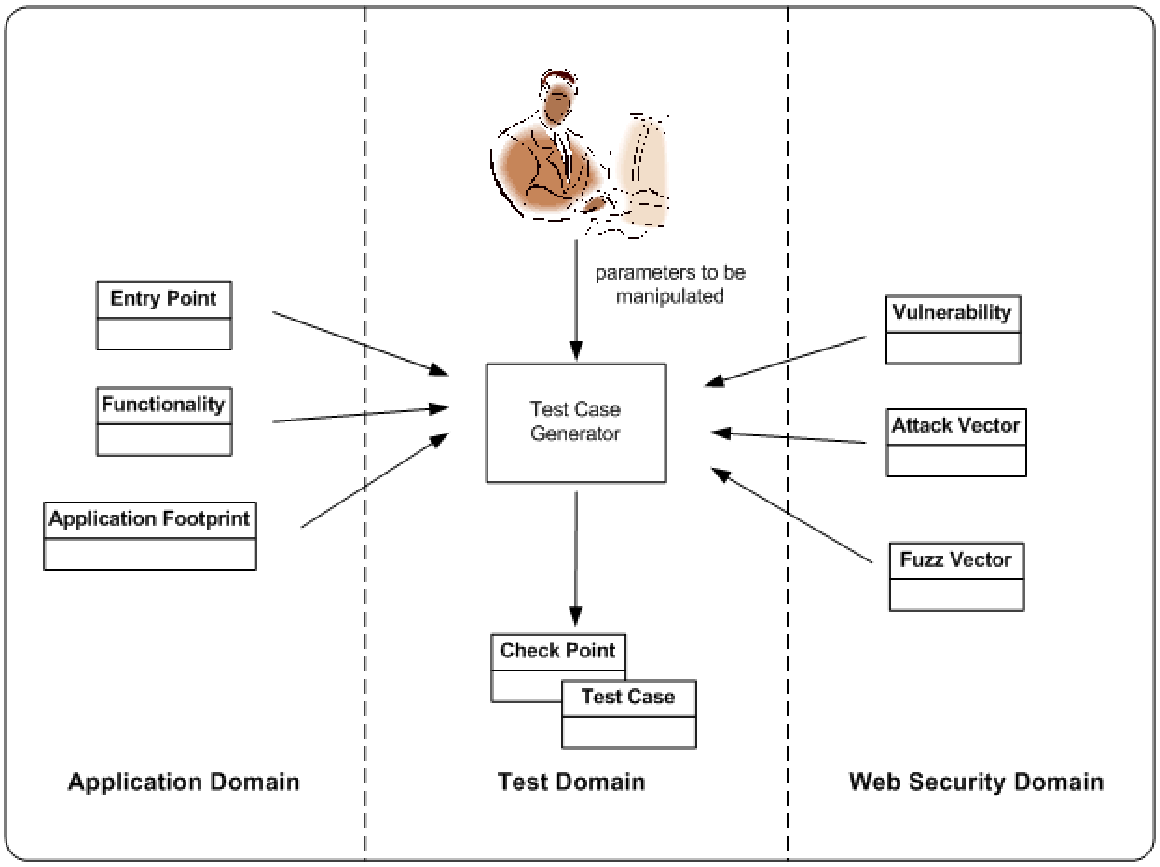


Figure 4 Model-Driven Penetration Testing

As shown in Figure 4, our proposed framework, with its supporting methodology and workbench will make possible a model-driven approach to penetration testing. Security artifacts, development artifacts, and test artifacts are represented in a declarative model that can be processed and supported by a variety of utilities and tools for managing a penetration test campaign. In particular, "Test Case Generator" utility can be used to generate test cases from explicitly modeled artifacts. A general tester works on the modeled artifacts and provides necessary application specific information to the Test Case Generator, e.g. tags an *Entry Point* with its associated functionality and marks the fields that will be manipulated to create attacks, and then uses the utility to process relevant security artifacts and development artifacts to generate concrete *Test Cases* automatically in a systematic consistent manner. All feasible *Attack Vectors* and appropriate *Fuzz Vectors* to construct each *Test Case* are automatically selected and applied.

Furthermore, since all the relevant artifacts in a test campaign are captured and retained using the model, they can be reused when test maintenance and regression testing are required.

4.2. Evaluation Criteria

The proposed test framework is intended to address the issues described in section 4.1. A careful literature review was conducted to identify the key criteria related to those issues. Then based on both the literature review, our analysis of the issues facing development teams in our case studies in Chapter 5, and discussions with both software engineering and security experts, a gap analysis was conducted to understand where the current approaches to penetration testing are problematic and deficient. Accordingly, the following set of evaluation criteria were identified that a framework for web application penetration testing should meet if they wish to address the issues outlined in section 4.1. The criteria are used in Chapter 6 to evaluate how well the proposed test framework improves on current practice for those particular issues. There may be other criteria that organizations and researchers will want to use, e.g. **Resultant Quality** which would measure whether or not the software created has better security using our framework and, **Return on Investment** (ROI) which evaluates the efficiency by quantitatively calculating the costs (e.g. learning curve, testing effort etc.) of the proposed framework versus that of current practices, in evaluating a test framework for penetration testing.

However, as the limitations mentioned in section 1.4, these criteria are outside the scope of our thesis. In this thesis we do not aim to do better than the quality of Expert Security Assessment (ESA), nor do we develop a rich enough toolset so that ROI can be evaluated in a major case study out in industry. Instead, we want to first focus on evaluating the framework in Chapter 6 against the criteria defined as below, and then we can justify any additional effort that will be required to address other constraints such as Resultant Quality and ROI.

Integrated Penetration Test Methodology

The test framework should provide a test methodology that guides internal testers on how to perform test campaigns throughout a security-oriented software development life cycle in collaboration with other members of the development team.

- Integrated Penetration Test Process: support a penetration test process that is completely integrated into a security-oriented software development life cycle
- Collaboration with Developers: facilitate collaboration between developers and testers throughout the security-oriented software development life cycle
- Development Artifacts Utilization: facilitate the use of development artifacts for testing purposes and provide feedback to developers on those artifacts
- Grey-Box Test Architecture: support penetration testing at any interface within a web application architecture using knowledge of the communications that take place between components
- Use with Other Security Test Methods: support the use of penetration testing in combination with other security test methods

Feasible for General Testers

General testers should be capable of leveraging the test framework to perform test campaigns in a cost-efficient approach and produce quality test results. They are not required to be a security expert and cannot be expected to have the knowledge and experience, on their own, to know all the types of vulnerabilities, attack vectors, and fuzz vectors that may be required for testing.

- Web Security Knowledgebase: provide a web security knowledgebase that retains and represents security expertise in a well-structured model that can be processed by computer programs and understood by general testers
- Test Campaign Modeling: test artifacts in a test campaign are identified and modeled so that they can be processed by computer programs, understood by general testers, and can be reused
- Workbench Tool Support: provide a PenTest workbench for general testers based on the web security knowledgebase and the test campaign model to support penetration testing at each phase
- Consistent Test Coverage: produce consistent test coverage with respect to entry points and types of vulnerabilities

Systematic Test Campaign Management

The test framework should provide guidance and tool support for systematic test campaign management, specifically, for test maintenance and regression testing.

- Reports and Documentation: provide a systematic approach to reporting progress in a test campaign based on well-defined reports and documentation
- Test Maintenance: support a systematic approach to test maintenance on a continuous base with respect to when and how to perform test maintenance
- Regression Testing: support a systematic approach to regression testing on a continuous base with respect to when to perform regression testing

4.3. Test Framework Overview

To be able to discuss and build up a tangible test framework in a feasible scale for our thesis research, we developed the proposed test framework based on three fundamental assumptions about software development process, application architecture, and web security knowledgebase:

Assumption 1: The development of web applications under test follows a classic iterative lifecycle process that follows a classic waterfall methodology within each iteration. This is typical of classic software development methodologies (e.g. RUP, the MS SDL) as discussed in section 2.3. In such a software development process, each iteration has three major phases: analysis, design, and implementation, and in each phase development artifacts such as functional specification, system architecture, API etc. are well documented and available to penetration testers. In addition, developers are available to assist in test efforts when required. This assumption has a direct impact on the test methodology of the framework.

Assumption 2: Web applications under tested using our framework are built based on a classic AJAX architecture. In other words, the web applications primarily process HTML-based information and may have rich client-side scripts, but they do not process multimedia content and do not interact with any plug-ins/add-ons in the browser that have security implications. This assumption has a direct impact on the grey-box test architecture, including the types and attributes of entry points and check points.

Assumption 3: For a penetration test campaign, security experts are available and relied on to build up and maintain the web security knowledgebase that is a critical component of the proposed test framework. The security experts ensure the web security knowledgebase is comprehensive and updated.

The proposed test framework consists of five major components, as shown in Figure 5, which will be discussed in detail in subsequent sub-sections:

- A **penetration test methodology** that provides guidance to testers for conducting a penetration test campaign that is integrated into a security-oriented software development life cycle. The test methodology defines the roles played by developers, penetration testers and security experts, and specifies the interactions between them in terms of the development artifacts utilized and the test artifacts produced in a test campaign.
- A **grey-box test architecture** that identifies the types of entry points and check points that exist across the complete architecture of a web application, that can be used while penetration testing.
- A **web security knowledgebase** which represents web security expertise about potential vulnerabilities in a web application using a well-structured format that can be processed by computer programs.
- A **penetration test campaign model** that captures the test artifacts, the development artifacts, and the web security artifacts in a test campaign throughout an entire security-oriented software development lifecycle including application maintenance. The test campaign model is leveraged to drive test development.
- A **knowledge-based PenTest workbench** that is built upon the penetration test campaign model and the grey-box test architecture to support the penetration test methodology using the web security knowledgebase. The PenTest workbench is designed to leverage various third party tools to facilitate test development and execution.

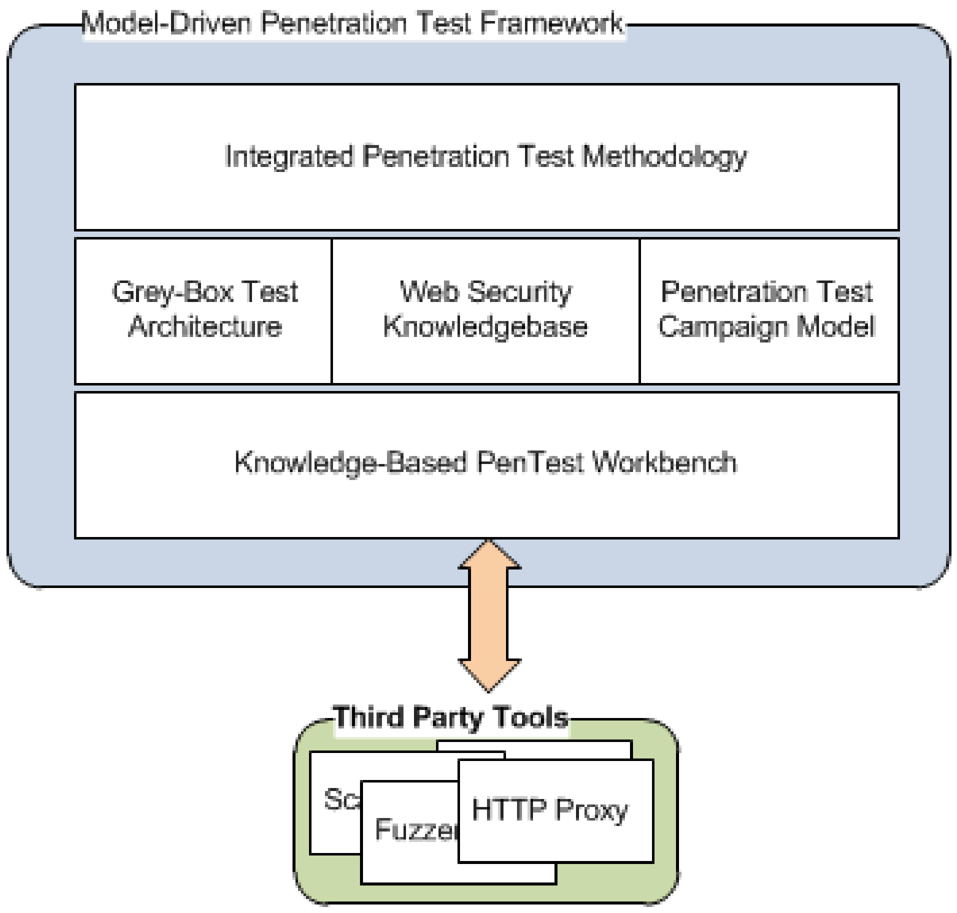

Model-Driven Penetration Test Framework
Integrated Penetration Test Methodology
Grey-Box Test Architecture
Web Security Knowledgebase
Penetration Test Campaign Model
Knowledge-Based PenTest Workbench
Third Party Tools
Sc
Fuzzer
HTTP Proxy

Figure 5 Test Framework Overview

4.4. Penetration Test Methodology

The penetration test methodology is designed to fully integrate penetration testing into a security-oriented software development life cycle. It provides guidance and support to an internal test team that initially does not have web security expertise built into the team.

4.4.1 Overview

Penetration testing is an ongoing activity throughout the entire security-oriented software development life cycle. Corresponding to the analysis, design and implementation phases in a typical software development life cycle, a penetration test campaign is an iterative process consisting of test analysis, test design, and test execution, as shown in Figure 6. New versions of test documents and other test artifacts are created or updated at the end of each phase in each iteration of the software development life cycle.

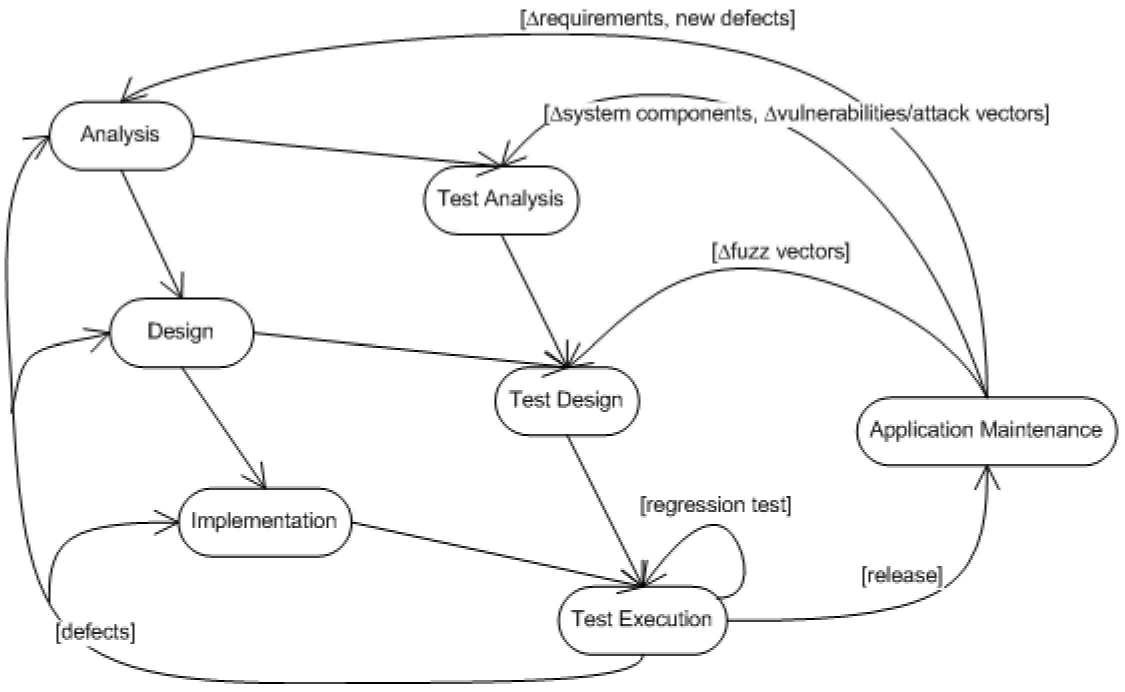


Figure 6 Iterative Software Development & Test Process

A penetration test campaign starts as early as the analysis phase of a software development life cycle. A Web Security Expert is invited to build up a web security knowledgebase for the web application, including identifying lists of *Functionality, Vulnerability, Attack Vector, and Fuzz Vector*. By utilizing application analysis artifacts and the web security knowledgebase, a Penetration Tester identifies the *Application Footprint* of the web application, and creates a list of *Potential Vulnerability* that potentially exists in the web application. The list of *Potential Vulnerability* is sent to a

Developer so that the security mechanisms for the vulnerabilities can be considered in the application design.

Once the test analysis is done and the application design starts, a Penetration Tester starts test design. By utilizing various application design artifacts and the web security knowledgebase, a Penetration Tester enumerates a list of *Entry Point* of the web application, generates a list of *Attack* and identifies *Check Point* for each *Attack*, and generates a list of *Test Case* to be executed.

At the test execution phase, when the web application is implemented and deployed to a test environment, the list of *Test Case* created at the test design phase are executed. A Penetration Tester analyzes the *Test Result* to confirm the identified vulnerabilities.

At the end of each test execution, any identified vulnerability defects are sent to a Developer to prioritize and fix. The defect fixes may result in changes in application implementation, design, or even analysis artifacts. Once the defects are fixed, test artifacts, e.g. *Test Case*, may need to be updated accordingly, and then a subset of or entire *Test Case* are re-executed to exclude any regression due to defect fixes.

When all the defects have been fixed and the application passes the final regression test, the application is released and the development and test process enters into the application maintenance phase. During the application maintenance phase, an application may be updated for various reasons, including new application or system requirements, new defects, and emerging vulnerabilities, attack vectors, fuzz vectors. As the result, test artifacts are updated accordingly, and a subset of or entire test cases are re-executed for security assurance.

Test documents to capture the test artifacts are created or updated at the end of each test phase.

4.4.2 Roles and Objectives

During a test campaign, a Penetration Tester works closely with an application Developer and a Web Security Expert to leverage development artifacts and security expertise.

The **Web Security Expert** builds up a web security knowledgebase at the beginning of the test analysis phase and transfers it to the Penetration Tester. Occasionally the Web Security Expert may re-visit the application development team to maintain and update the web security knowledgebase. At the end of the test execution phase, the Web Security Expert can also review test campaign artifacts and test results with the Penetration Tester and the Developer.

The **Developer** provides relevant development artifacts and application knowledge at each test phase for test analysis and test case design. At the end of the test analysis phase, the Developer takes a list of Potential Vulnerability from the Penetration Tester and utilizes it in the design of the web application security mechanisms. During the test design phase, the Developer assists the Penetration Tester with identifying appropriate check point(s) for each attack. At the end of the test execution phase, the Developer is responsible for addressing all the identified vulnerabilities. The term

Developer represents various members on a development team, such as system analyst, designer, and programmer.

The **Penetration Tester** is responsible for managing a test campaign, including the test development by utilizing development artifacts and web security knowledgebase, the test execution, the delivery of test result (report), and the test maintenance of all relevant artifacts. The Penetration Tester also ensures that the Web Security Expert and the Developer provide all required inputs and addresses all feedback provided.

4.4.3 Test Analysis

As shown in Figure 7, there are five basic activities occurring during the test analysis phase:

System Analysis

The Developer performs system analysis for an application under development and produces two development artifacts: a list of *Use Case* that identifies what main interactions the web application must support and the *Functional Specification* that describes in detail the functionalities provided by the web application.

Build Up Web Security Knowledgebase

The Web Security Expert is engaged to build up a web security knowledgebase for the web application. The web security knowledgebase includes structured descriptions of all the types of *Vulnerability* and related *Functionality*, *Attack Vector,* and *Fuzz Vector* that are relevant to the web application.

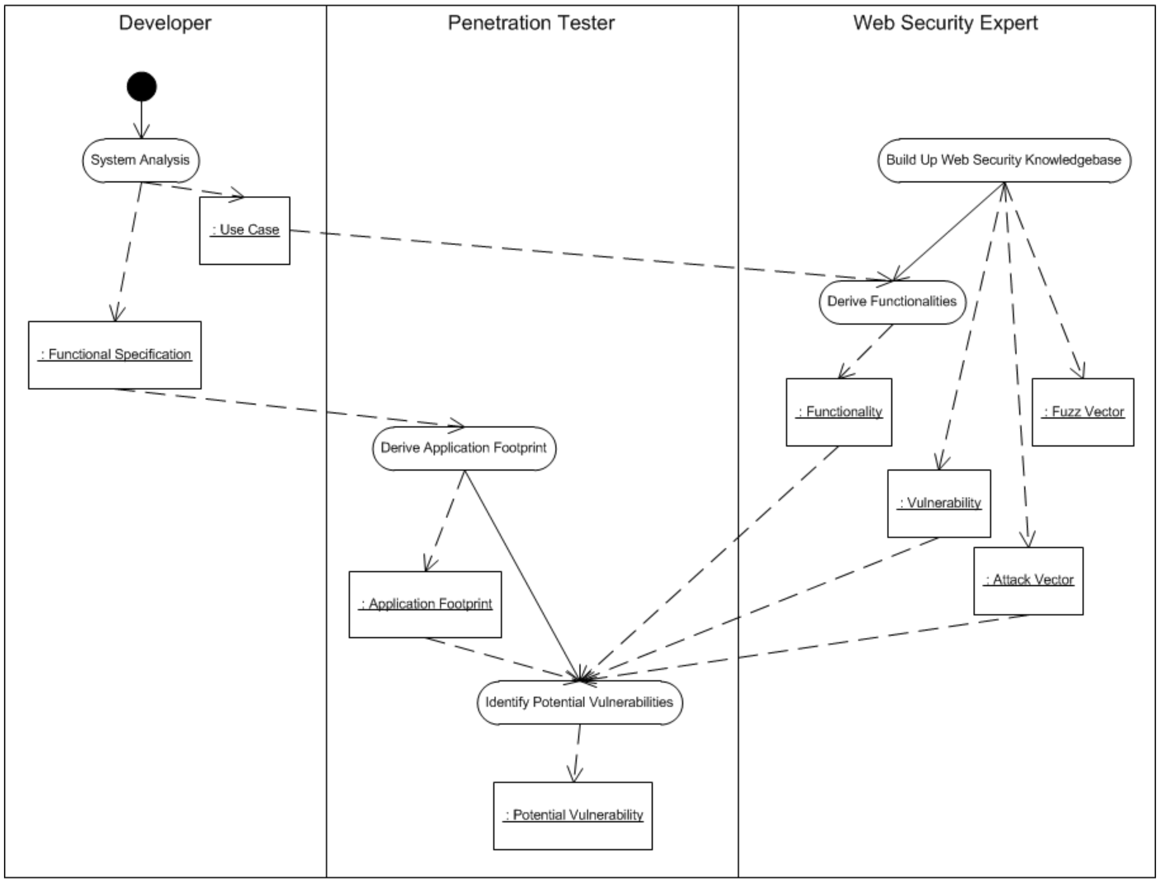


Figure 7 Methodology: Test Analysis

Derive Functionalities

The Security Expert identifies and categorizes the list of *Functionality* that the web application provides based on the analysis of the list of *Use Case* provided by the Developer. This list is used to identify the list of *Potential Vulnerability*. Penetration Tester will refer to the functionalities specified in the security knowledgebase later on when they enumerate entry points.

Derive Application Footprint

The Penetration Tester derives the platform and configuration details relevant to the web application from the *Functional Specification* and produces the test artifact

Application Footprint. This is used to more precisely identify a platform specific list of *Potential Vulnerability*.

Identify Potential Vulnerabilities

The list of *Functionality* and *Application Footprint* are used by the Penetration Tester to index into the web security knowledgebase to identify the list of potential *Vulnerability* and *Attack Vector* that exist in the application, and produces the test artifact: list of *Potential Vulnerability*.

```
Document: Phase I Penetration Test Analysis
Project:
Author:
Date:

1. Purpose

2. Development Artifacts

3. Application Footprint
   Platform
   Components
   Database
   Operating System
   Web Server
   Encoding

4. Potential Vulnerabilities with related application Functionalities
```

Functionality	Vulnerability	Attack Vector	Description

Figure 8 Test Analysis Document Template

At the end of the test analysis phase, the Penetration Tester produces a test analysis document that compiles all the created test artifacts at this phase and lists the development artifacts that are utilized. Figure 8 shows a simple template for such a test analysis document which includes a list of the utilized development artifacts and the description of the test artifacts including *Application Footprint*, *Functionality* and

Potential Vulnerability identified at the test analysis phase. Appendixes A1, B1, C1 present the test analysis documents for the three case studies described in Chapter 5.

4.4.4 Test Design

The test design phase is coordinated with the application design phase. At the beginning of the application design phase, as shown in Figure 9, the list of Potential Vulnerabilities identified at the test analysis phase is sent to the development team so they can design appropriate countermeasures into the web application.

There are five basic activities occurring during the test design phase that are discussed in detail below:

System Design

The Developer performs system design and produces three development artifacts: list of *Web API* that defines a set of HTTP request messages along with the descriptions of parameters and response messages, *System Architecture* that specifies the structure of a web application in terms of sub-systems and/or components and the interactions between the sub-systems and components, and list of *User Scenario* that describes the interactions between end users and the web application.

Enumerate Entry Points

A Penetration Tester enumerates all entry points and identifies which functionality, from the list of *Functionality* created at the analysis phase, is provided by each entry point. The test artifact *Entry Point* is created which defines precisely how the entry point is invoked, and what arguments are passed in, in terms of the *Web API*.

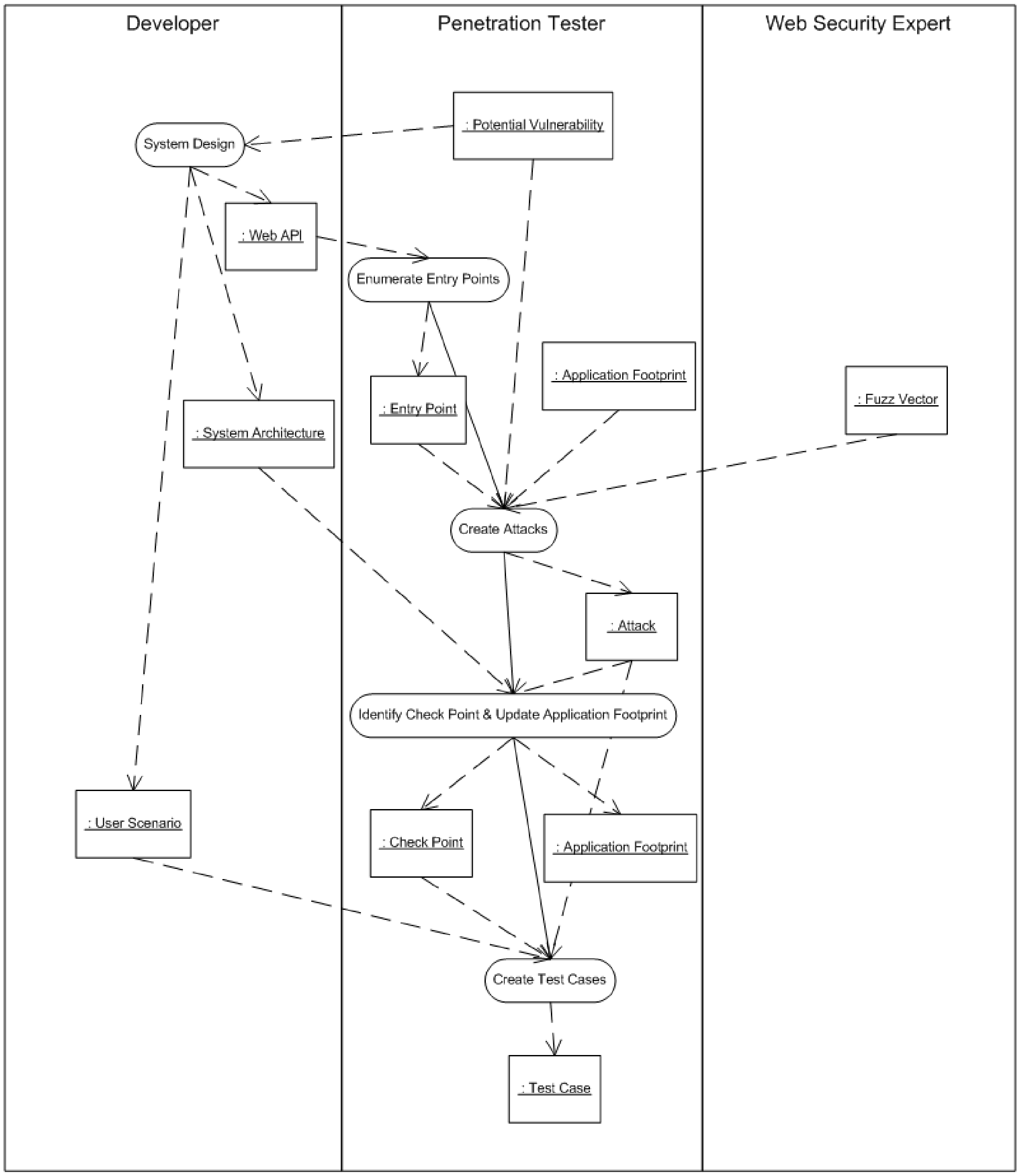


Figure 9 Methodology: Test Design

Create Attacks

Based on the functionality of each entry point (any user input that can be used to attack a web application, as defined in section 1.5) and the list of *Potential Vulnerability* identified at the test analysis phase, a Penetration Tester looks up *Fuzz Vector* in the web security knowledgebase that are relevant to the *Application Footprint* to create list of *Attack* that will be used to generate list of *Test Case*.

Identify Check Points & Update Application Footprint

For each attack, based on the development artifact *System Architecture* and the interaction with a Developer, a Penetration Tester identifies *Check Point*(s) with expected result(s) that will be used to determine if the attack succeeds or not. A Penetration Tester produces the test artifact *Check Point* at this phase. A Penetration Tester may also update the *Application Footprint* to identity which application footprint components are intended to provide built-in protection against vulnerabilities. For example, if the web application is using the .Net Framework there are some built-in filters that protect against some basic XSS attacks.

Create Test Cases

Based on the created list of *Attack* with the identified *Check Point*(s) that provide functionality corresponding to a *Use Case*, the Penetration Tester chooses an appropriate user scenario from the development artifact *User Scenario* to create executable test cases, and produces the test artifact: list of *Test Case*.

At the end of the test design phase, the Penetration Tester produces a test design document that compiles all the created and updated test artifacts at this phase and lists the development artifacts that are utilized. Figure 10 shows a simple template for such a test design document which includes a list of the utilized development artifacts and the description of the test artifacts including *Application Footprint*, *Entry Point*, *Attack* and

Check Point, and *Test Case*. Appendixes A2, B2, C2 present the test design documents that were created for our three case studies that are described in Chapter 5.

```
                    Document: Phase II Penetration Test Design
                                    Project:
                                    Author:
                                     Date:

1. Purpose

2. Development Artifacts

3. Application Footprint

4. Entry Points with Attacks & Check Points
--------------- List of Entry Points --------------
# of entry points:
# of attacks:

Entry point: <Page name> (ID: <ID>)
GET <URL>
Functionality: <One of Navigation, Log On etc.>

Entry point: <Page name> (ID: <ID>)
POST <URL>
Form Fields:
   <list of field names>
Functionality: <One of Navigation, Log On etc.>
Vulnerability: <List of possible vulnerabilities>
Attack: <ID>
  Attack Description: <Description of the attack>
  POST <URL>
  Manipulated Form Field: <Field to be manipulated>
  Defense Mechanism: <Type of defense mechanism, e.g. Application Specific,
APS.net etc.>
  Check Point <ID>: <Location of the check point, e.g. Page Name, Application
Log file etc.>
    Expected Result:
-----------------------------------------------------



5. User Scenarios
--------------- List of User Scenarios --------------
# of user scenarios:

User Scenario <ID>: <Description of the user scenario>
  Step <ID>: Entry Point: <Name of the entry point> (ID: <ID>)
  Description: <Description of the step>
-----------------------------------------------------



6. Test Cases
--------------- List of Test Cases --------------
# of test cases:

Test Case: <ID>
Scenario <ID>:  <Name of the scenario>
  Step <ID>: GET/POST <URL>
  Step <ID>: Attack <ID>
-----------------------------------------------------
```

Figure 10 Test Design Document Template

4.4.5 Test Execution

As shown in Figure 11, there are three basic activities occurring during the test execution phase that are discussed in detail below:

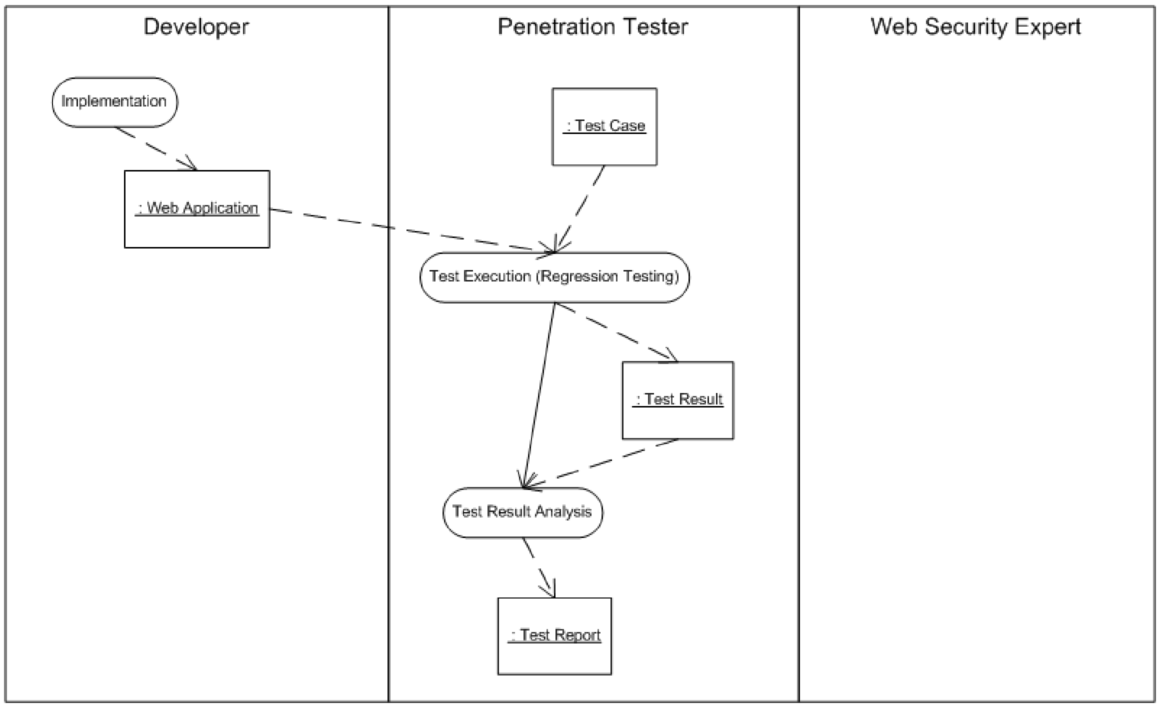


Figure 11 Methodology: Test Execution

Implementation

The Developer implements the application design and builds an application deliverable that is deployed in the test environment. At least one application deliverable is built and deployed for each iteration of the software development lifecycle.

Test Execution (Regression Testing)

The Penetration Tester executes the list of *Test Case* created in the test design phase and produces the test artifact *Test Result* for each test case executed. During this phase, many, if not all, of the test cases will need to be re-executed a few times to follow up on bug fixes. A complete re-execution of all test cases (regression testing) will be done once at the end of the phase to ensure that there were no side effects that broke tests which had previously been successful.

Test Result Analysis

The Penetration Tester reviews the test results with Developer to analyze failed test cases to confirm the existence of uncovered vulnerabilities. There are four possible results of test review: True Negative, False Negative, True Positive, False Positive (as defined in section 1.5).

If an uncovered vulnerability does exist (**True Negative**), a Developer needs to evaluate its exploitability and risk, and take appropriate countermeasures, e.g. fix the flaws in application, to remedy the vulnerability. The remediation measures may need to be verified by a Security Expert (internal or external) to make sure they are adequate and appropriate. The fix may result in changes in application implementation, system design, or system analysis, as shown in Figure 6. Consequently, a Penetration Tester may need to perform test analysis and test design to update existing test artifacts such as *Potential Vulnerability*, *Entry Point*, *Test Case* etc., and then execute the updated test cases to validate the fix in addition to the impacted test cases for regression; or simply re-execute the updated test cases to validate the fix.

If an uncovered vulnerability does not exist (**False Negative**), a Penetration Tester will work with a Developer to improve the design of the checkpoint so that the test result can be verified more precisely. Usually a False Negative means that the application is blocking the attack appropriately, but it is not logging or returning a result that would enable a test to confirm for sure that the attack was blocked. Consequently, the test case may need to be updated accordingly, and it will need to be re-executed to ensure the false negative has been eliminated.

Usually, during test review Penetration Tester and Developer do not examine any passed test case in order to confirm that it is a "**True Positive**" (which means a test correctly confirmed that an attack was blocked and there is no need for any further action). In general, it would be hard in our framework to identify a "**False Positive** (which means a test seemed to confirm that an attack was blocked, when it was not).However, in case a "False Positive" is identified by any chance during the test result review, a Penetration Tester needs to figure out why the security defect is not uncovered in the test campaign (for example, it may be due to a coding error at either the entry point or the check point. A test result is logged indicating that the attack was blocked when in fact it succeeded. Or it could be that the set of fuzz vectors is not sufficient. All tests passed, but there was a different fuzz vector that could exploit the vulnerability that was not tried). Then a correction action is taken accordingly, e.g. fix the entry point or check point code, or update the web security knowledgebase by replenishing new vulnerability information (in this case, a Security Expert may be required to get involved). New test case(s) or updated test case(s) is generated afterwards, and then is executed to make sure the security defect is uncovered. The test case(s) will then be re-executed when the defect is fixed.

Based on the test result analysis, a Penetration Tester creates the test artifact *Test Report* to summarize the results of test execution.

At the end of the test execution phase, a Penetration Tester produces a test execution document that compiles all the created test artifacts at this phase. Figure 12 shows a simple template for such a test execution document which includes the application under test and the version information and the description of the test artifacts including *Test Result*, and *Test Report*. Appendixes A3, B3, C3 present the test execution documents that were created for our three case studies that are described in Chapter 5.

```
                 Document: Phase III Penetration Test Execution
                                   Project:
                                   Author:
                                    Date:

1. Purpose

2. Test Result Summary
2.1 Application under Test:

2.2 Test Report
------------- Test Campaign Summary --------------
# of test cases executed:
# of failed tests:
# of identified vulnerabilities:
# of entry points tested:
# of vulnerable entry points:
--------------------------------------------------

------- Web Security Repository Statistics -------
# of vulnerabilities:
# of attack vectors:
# of fuzz vectors:
--------------------------------------------------

----------------Test Result Details---------------
Test Case <ID>: <Test Case description>
Result: <Pass/Failed>
--------------------------------------------------

------- Identified Vulnerability Details ---------
Vulnerability: <Description>
probably exists on:
  Entry Point: <ID>
--------------------------------------------------
```

Figure 12 Test Execution Document Template

4.4.6 Application Maintenance

During the application maintenance phase, an application may change for a variety of reasons, such as adding new features, fixing reported defects, and upgrading third-party system components. In addition, any updates to the web security knowledgebase may require test artifacts be updated. These changes may require re-running the test campaign, as shown in Figure 6:

- Δ **Requirements**: when new features are added in an application this can result in new entry points (and associated functionalities), as well as new use cases and user scenarios etc. While a Developer needs to go through the whole development process, a Penetration Tester will need to go through the process of Test Analysis/Test Design/Test Execution as well. Consequently, new *Potential Vulnerability* may be identified, new *Entry Point* enumerated, and new *Test Case* created. New test cases, in addition to the whole or a subset of existing test cases, are executed and the list of *Test Result* is analyzed and reported.
- **New Defects**: when new defects are fixed during the application maintenance phase, a Developer goes through the same process as in the test execution phase. Depending on the impact of the defect fix, a Developer will need to perform application analysis, or design or implementation. Consequently, a Penetration Tester will need to perform test analysis and test design to create new test artifacts of such as *Potential Vulnerability*, *Entry Point*, *Test Case* etc., and then execute the updated test cases plus the impacted test cases for regression to validate the fix; or simply re-execute the impacted test cases to validate the fix.
- Δ **System Components**: when the third-party components used in an application change, e.g. a new third-party component is used or an existing third-party component is upgraded, the application footprint may change. As a result, a Penetration Tester will need to perform test analysis and test design to

identify new potential vulnerabilities, or new feasible fuzz vectors which will be used create new test cases. In addition, a Penetration Tester may need to update the existing check points due to changes in the third-party component. All these require updates in the test artifacts such as *Potential Vulnerability*, *Attack*, *Check Point*, *Test Case* etc. The updated test cases, in addition to a subset of existing test cases, need be re-executed for regression.

- Δ **Vulnerabilities/Attach Vectors**: when the web security knowledgebase is updated with new vulnerabilities or attack vectors, a Penetration Tester needs to perform test analysis to identify new potential vulnerabilities and/or attack vectors. Consequently, a Penetration Tester may update the test artifacts of *Potential Vulnerability*, *Attack*, and *Test Cases*. The new test cases are executed for regression.
- Δ **Fuzz Vectors**: when the web security knowledgebase is updated with new fuzz vectors, a Penetration Tester needs to perform test design to create additional attacks. Consequently, the Penetration Tester may update the test artifacts of *Attack*, and *Test Cases*. The new test cases are executed for regression.

4.5. Grey-Box Test Architecture

In a security-oriented software development life cycle, testers can take a grey-box approach to penetration testing by taking advantage of their access to development artifacts, application components, and developers' expertise. The grey-box test approach provides greater flexibility of identifying and defining entry points and check points. In the following sub-sections, we describe in detail how a grey-box test architecture integrates well with web application architecture, and give a complete description of entry points and check points.

4.5.1 Web Application Architecture

Figure 13 shows the architecture of a typical AJAX web application.

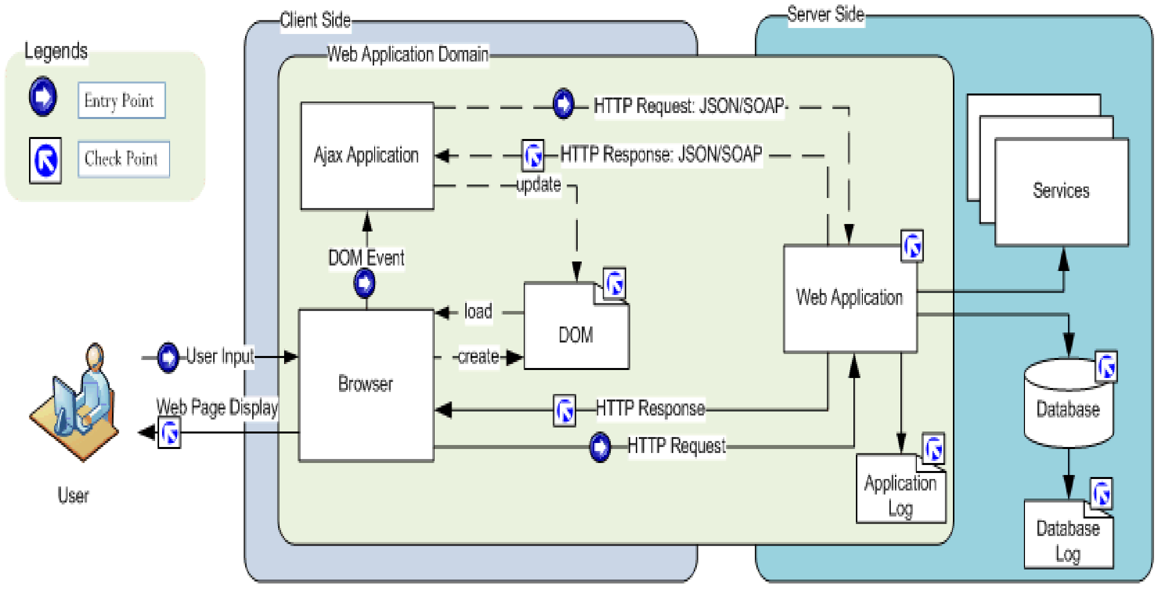


Figure 13 AJAX Web Application Architecture

At the client-side, a user interacts with a web application via a Browser, which supports User Input and a Web Page Display. There are two types of HTTP traffic between the Browser and the Web Application. The solid line shows the Browser directly sending an HTTP request to the Web Application and then receives an HTTP response from the Web Application that is used to refresh the Web Page Display. The dashed line shows that User Input to the Browser triggers a DOM event that in turn triggers an AJAX script to send a JSON/SOAP message to the Web Application and then receive a JSON/SOAP message back from the Web Application. Using the received JSON/SOAP message, another AJAX script updates the DOM that is then loaded into the Browser to update the Web Page Display.

At the server-side, the Web Application saves and retrieves persistent data in the Database. The Web Application interacts with the Services to delegate certain functionalities, which can run locally on the same server or run remotely on a different server. Both the Web Application and the Database produce logs, the Application Log and the Database log, that can be used as check points to verify if an attack succeeds or not.

Developing such a grey box test architecture with visibility into the entire application architecture is advantageous to penetration testing as it gives a rich set of entry points and check points that can be used for constructing test attacks and verify the test results. These are described in the following sections.

4.5.2 Entry Point

An *Entry Point* is any place where a potential hacker can interface with the web to launch an attack against a web application. Like a hacker, a penetration tester can launch attacks against a web application at the client-side from various points. An attack can be launched simply by User Input via Browser, or by DOM Event via DOM inspector, or by directly manipulating HTTP requests via a HTTP proxy. The HTTP requests can be a standard HTTP request ("HTTP Request" in Figure 13) sent out by Browser, or specific HTTP request such JSON or SOAP message ("HTTP Request: JSON/SOAP" in Figure 13) sent out by an AJAX Application.

In Figure 13, the points with right arrow are the potential entry points, including User Input on a web page in browser, DOM Event, or HTTP Request message (standard

HTTP request message or JSON/SOAP message). Table 1 shows a comparison of the three types of entry points.

Table 1 Comparison of Entry Points

User Input	
advantage	1) Easy to identify and to launch attacks 2) Intuitive to a general penetration tester
disadvantage	1) Limited types of attack surface: only URL and input fields 2) Difficult to manipulate hidden fields 3) Attacks are constrained by GUI controls and client-side validation
DOM Event	
advantage	Can launch all types of DOM related attacks
disadvantage	Need tool support, e.g. DOM inspector
HTTP Request (standard HTTP request message or JSON/SOAP message)	
advantage	It may provide the most flexibility in attack manipulation: 1) All types of attack surface except client-side DOM attack 2) HTTP protocol 3) Arbitrary encoding schema
disadvantage	1) Need tools support, e.g. HTTP proxy 2) Not intuitive to a general penetration tester

User Input entry points are straight forward to identify if they are specified in a design document such as what we have referred to in our thesis as the Web Application API. It is also straight forward to use a Browser or simulation tools to launch attacks. It is the most intuitive approach for a penetration tester. However, it has limited capabilities to manipulate attacks.

DOM Event entry points is the best choice for manipulating attacks related to DOM, including those client-side DOM attacks that do not generate HTTP traffic from the client to the server. Tool support is required in order to manipulate entry points and launch attacks.

HTTP Request entry points may provide the most flexibility in attack manipulation. Tool support is needed to manipulate entry points and launch attacks. Since the entry points are presented in the format of HTTP request messages, it is not intuitive to a general penetration tester to interpret and manipulate such entry points.

4.5.3 Check Point

A *Check Point* is any information that can be used to verify if an attack succeeds or not. A penetration tester who uses a grey-box test architecture has more flexibility to choose check points that can be at the client-side and/or at server-side. Choosing an appropriate check point(s) for an attack is determined by the nature of an attack, e.g. where the attack occurs – at the server-side or at the client-side, and the nature of the application component that is implemented to defend against the attack.

As shown in Figure 13, the points with the left-up arrow are the potential check points, including the content on Web Page Display in Browser, the content in DOM, HTTP Response message ("HTTP Response" and "HTTP Response: JSON/SOAP" in Figure 13) at the client-side, and the states in the Web Application memory, the content in the Database, the entries in the Application Log or the Database Log at the server-side. Table 2 shows the comparison of the various types of check points.

Table 2 Comparison of Check Points

Client Side	
Web Page Display	
advantage	Intuitive, and is reliable to check the attacks that occur at the client-side, e.g. all types of XSS.
disadvantage	May need to run a test against all brands and versions of web browser if a vulnerability is browser-specific.
DOM	
advantage	Suited to verify the attacks that exploit client-site scripting.
disadvantage	Less intuitive, and need a DOM inspector to verify check points.
HTTP Response (standard HTTP response message or JSON/SOAP message)	
advantage	Easy to build a utility to intercept and analyze response messages as there are many HTTP API available.
disadvantage	1) For some types of attacks that occur at the client-side and are interrelated to specific browsers, e.g. XSS, only checking HTTP responses may result in false negative or false positive. 2) Cannot detect attacks that do not involve HTTP communications, e.g. DOM-based XSS.
Server Side	
Web Application memory	
advantage	Can observe dynamic behavior of attacks.
disadvantage	It may be complicated to build a utility to monitor and check the states in server memory.
Database	
advantage	Can reliably check the permanent data changes caused by attacks.
disadvantage	May need to build a utility to parse and check records in database.
Application Log & Database Log	
advantage	Can focus on those abnormal events and exceptions that are usually related to malicious attacks.
disadvantage	May need to build a utility to parse and check log data.

For each check point, an expected result (oracle) should be specified. The expected result can be a general oracle that adheres to an attack vector, e.g. "Bypass Logon authentication" is a general oracle for a successful SQL Injection attack against "Logon" functionality. A general oracle can be vague and less precise, as it does not consider the context of a specific application. Application-specific expected results are more precise so that they can be utilized to reduce false positives and false negatives in test results, although extra efforts are required to identify and specify application-specific expected result for a check point.

Penetration testers work with developers to utilize their expertise in an application domain to identify and specify check points and expected results. This collaboration process is a good opportunity for both the developers and the testers to get a better understanding of how defense mechanisms in an application work, and therefore where and what to check to determine if any attack has been trapped and reported.

4.6. Web Security Knowledgebase

Security experts bring web security expertise into an application development project. In order for an organization to effectively use that knowledge, and retain it after the experts have left, it is important to be able to declaratively represent their knowledge in a form that can be processed by computer programs. In the proposed test framework, a web security knowledgebase is proposed to achieve this. The completeness and consistency of the web security knowledgebase is a critical element in determining the level of security provided by the test framework. In the following subsections we detail the expert knowledge that is relevant to penetration testing in terms of attack anatomy, attack vectors and fuzz vectors, and then introduce the model used to represent knowledge in our web security knowledgebase.

4.6.1 Attack Anatomy

The purpose of penetration testing is to ensure that the web application cannot be compromised by an attack. In general, launching an attack involves three security artifacts: vulnerabilities, attack vectors, and fuzz vectors.

Vulnerability

A vulnerability is a security defect in a web application. It exists in an application's code at either server-side or client-side. For example, Stored XSS, Reflected XSS, and SQL Injection are types of vulnerabilities that can exist in server-side application code, while DOM-based XSS and DOM-based injections are types of vulnerabilities that can exist in client-side application code.

Attack Vector and Fuzz Vector

An attack vector is a scenario of interaction in which a hacker can launch an attack through an entry point against an application. A fuzz vector is a specific value that can be used to parameterize or replace part of an entry point as a malicious payload in an attack. Attacks are actions to exploit the vulnerabilities in a web application by loading malicious payloads. While an attacker always launches an attack on the client-side to exploit a vulnerability, the attack may occur at either server-side or client-side. The attack does not necessarily occur at the same place where the vulnerability exists. For example, SQL Injection attacks occur at the server-side where also the vulnerability exists, while all XSS attacks occur at the client-side no matter where the vulnerability exists. An attacker can attack against a web application directly, e.g. SQL Injection attack, or against a legitimate user of the web application, e.g. XSS attack.

4.6.2 Characteristics of Attack Vectors

An analysis of attack vectors documented in (OWASP TOP 10, 2007), (SANS TOP 20, 2011), and (WASC, 2010), reveals the following characteristics:

- Many attacks are performed via "URL manipulation" – manipulating HTTP requests including cookie, URL parameter or path, and form fields.
- Although many attack vectors are applicable to any web application platform, there are some attack vectors which target specific platforms or components. For example, PHP applications are particularly vulnerable to "Malicious File Execution".
- Most attacks are based on deliberately misusing the most common functionalities of a web application. Therefore, the functionality of an entry point is a key indicator to identify potential attack vectors. For example, "search" is very likely vulnerable to reflected XSS attacks; in a public forum, "viewing" contents that were entered by other users is vulnerable to stored XSS attacks; and password-based "Login" is vulnerable to SQL Injection attacks.

4.6.3 Characteristics of Fuzz Vectors

An analysis of fuzz vectors documented in (OWASP Testing Guide, 2008), (Hansen, 2011), (Daw, SQL Injection Cheat Sheet, 2009), and (Daw, Input Validation Cheat Sheet, 2009), reveals the following characteristics:

- Fuzz vectors are categorized by attack vectors, e.g. fuzz vectors for XSS, fuzz vectors for Buffer Overflows, and fuzz vectors for Format String Errors.
- Some fuzz vectors are database specific. For example, fuzz data of SQL Injection for Oracle, SQL Server, and MySQL might be different.
- Some fuzz vectors are OS specific. For example, on a Unix OS, root directory is "/", and directory separator is "/", while on a Windows OS, root directory is "<drive letter>:\", and directory separator is "\" but also "/".
- All fuzz vectors may be obfuscated using different encoding schemas. This is an efficient mechanism to bypass filtering/sanitization defenses in web applications.

4.6.4 Penetration Test Security Model

Based on the characteristics of the security artifacts (in the color of orange) – *Vulnerability*, *Attack Vector*, *Fuzz Vector* and *Functionality*, and the test artifacts (in the color of blue) – *Entry Point* and *Application Footprint*, a penetration test security object model that captures key entities in each artifact and the relationships between the artifacts was created, as shown in Figure 14:

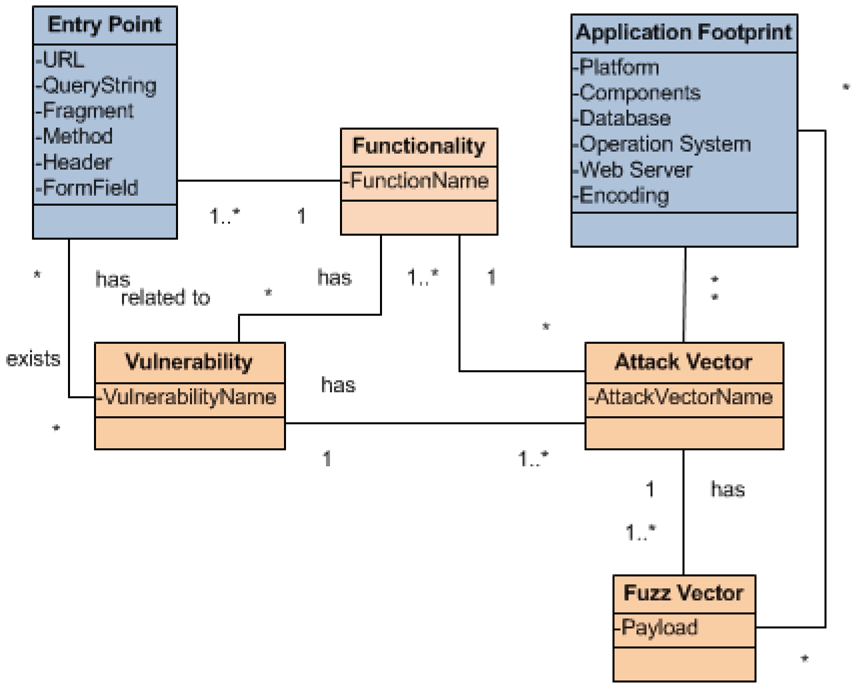


Figure 14 Web Application Penetration Test Security Model

- **Entry Point**: a test object that consists of the attributes that represent the possible fields that can be manipulated to construct an attack. It has six attributes – *URL*, *QueryString*, *Fragment* that marks a location on a web page, *Method* that is an HTTP method used in this entry point, *Header* that is the header of an HTTP message, and *FormField* that is the name of the input field in a form. An entry point may have one or more vulnerabilities or no vulnerability at all. An entry point performs a particular functionality.

- **Application Footprint**: a test object that consists of the attributes that characterize a web application. It is used to select relevant web security objects such as attack vector and fuzz vector and identify appropriate check points. It has six attributes – *Platform* that is the language and running environment on which a web application is developed and runs, *Components* that specifies third-party security components used in a web application that defense attacks and therefore can be used to check if a test attack succeeds or not, *Database* that is the database used by the web application, *Operating System* on which the web application runs, *Web Server* on which the web application runs, and *Encoding* that is the encoding schema used by the web application. A particular application footprint may be related to many attack vectors and fuzz vectors.
- **Vulnerability**: a web security object that represents a web application vulnerability. A vulnerability is related to one or various types of functionalities. A vulnerability has one or more attack vector(s).
- **Functionality**: a web security object that represents a function can be used to characterize vulnerabilities that potentially exist in a web application. A Web Security Expert creates such vulnerability-related functionalities during the setup of a web security knowledgebase for a web application. A functionality may exist on one or many entry points. A functionality may has many potential vulnerabilities and attack vectors.
- **Attach Vector**: a web security object that represents an attack vector. An attack vector is applied via performing a function. An attack vector may be applied on applications with many different footprints. An attack vector has one or more fuzz vector(s).
- **Fuzz Vector**: a web security object that represents *Payload* that is used to manipulate a specific field in an entry point. A fuzz vector may be applied to applications with many different footprints.

This object model can be used to implement a web security knowledgebase in a variety of different repositories, e.g. a relational database, an object database, or an XML database.

4.7. Penetration Test Campaign Model

The web security knowledgebase is just one input into the process of penetration testing. As shown in the figures that document the penetration test methodology in sections 4.4.1, 4.4.3, 0, and 4.4.5 there is a larger set of artifacts that must be maintained and managed throughout a penetration test campaign. As with the web security knowledgebase, it is advantageous if these artifacts can be declaratively represented in a well-structured format that can be processed both by computer programs and by developers and testers. Figure 15 shows the penetration test campaign model that we have built to manage and maintain a test campaign. In particular we have classified the objects in this model into three domains: web security domain (corresponds to the web security knowledgebase) which is maintained by the Web Security Expert, a development domain which is managed by the Developer, and the penetration test domain which is the responsibility of the Penetration Tester. We have further classified objects in the penetration test domain as to which test phase (test analysis, test design, or test execution) when they are created.

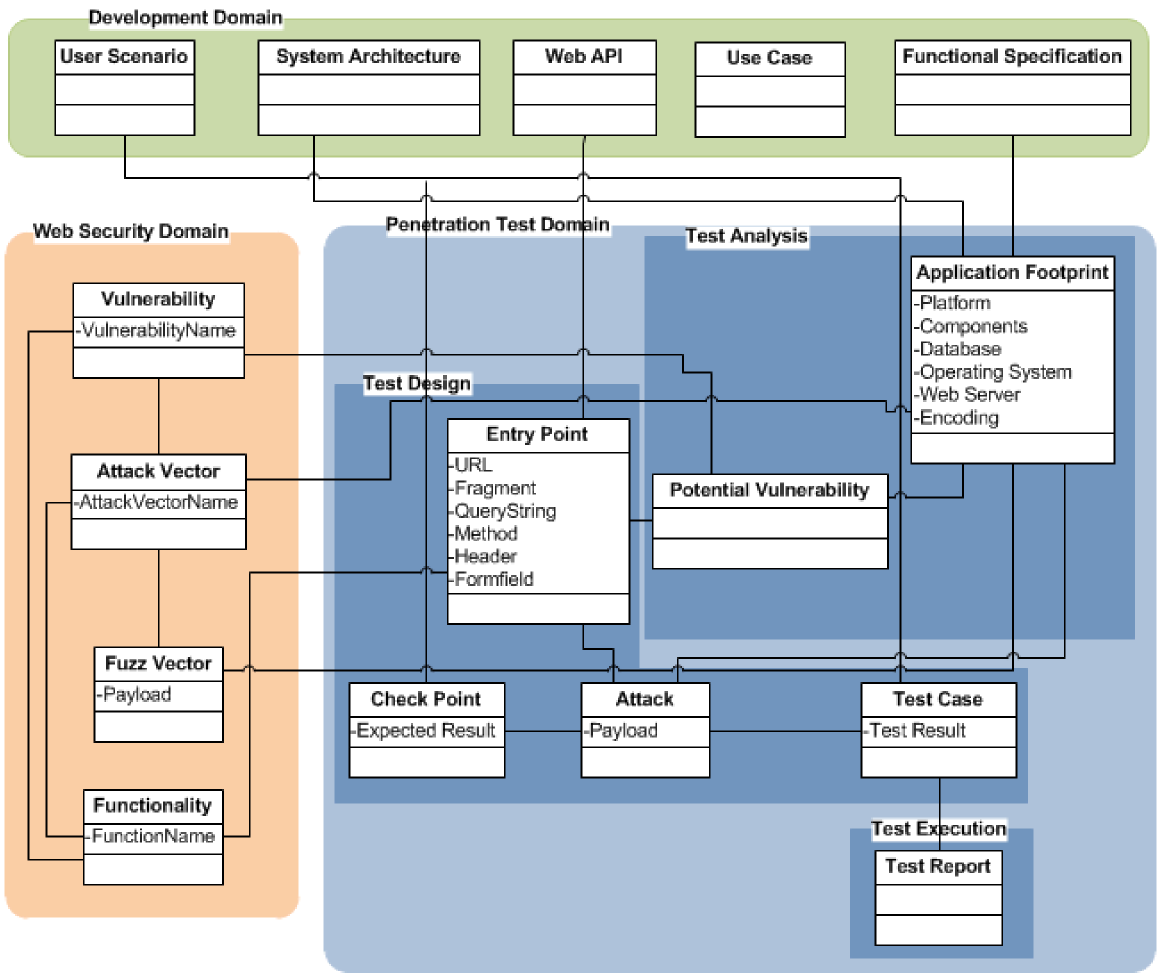


Figure 15 Penetration Test Campaign Model

In a test campaign, the Penetration Tester follows the test methodology proposed in section 4.4 building and interacting with the penetration test campaign model to manage the test efforts:

- **Test Analysis Phase.** At the test analysis phase, a Web Security Expert is invited to the project team to build up a web security knowledgebase that is relevant to the application based on the characteristics of the application e.g. its functionalities. The web security knowledgebase includes four artifacts: *Vulnerability*, *Attack Vector*, *Fuzz Vector*, and *Functionality*. A Penetration Tester starts to create the test artifacts *Application Footprint* based on the development artifact *Functional Specification*. Then based on the application functionalities and footprint, a set of potential vulnerabilities that may exist in the application can be identified and the results are saved as test artifact *Potential Vulnerability*.

- **Test Design Phase.** At the test design phase, a Penetration Tester collects a complete list of *Entry Points* based on the development artifact *Web API*, and identifies the functionality for each entry point. Based on the functionality, potential vulnerabilities can be identified, and for each of them a set of attacks are created by applying viable attack vectors and fuzz vectors with respect to the application footprint to manipulate the entry point. The results are saved in the test artifact *Attack*. For each attack, based on the development artifact *System Architecture*, a Penetration Tester works with a Developer to identify one or more check points and saves the results in the test artifact *Check Point*. Based on the discussion with a Developer during the process of identifying check points, a Penetration Tester may need to update the "Components" field in the *Application Footprint* to specify any component in the application that plays a security role to defend one or more types of attacks. At the end, for each attack, a viable user scenario that has the entry point corresponding to the attack is selected based on the development artifact *User Scenario*, and then a test case is created based on the user scenario and the attack. The results are saved in the test artifact *Test Case*.
- **Test Execution Phase.** At the test execution phase, a Penetration Tester executes the created test cases and saves the *Test Results*. The Penetration Tester works with a Developer to analyze the test results to identify any false-positives. A *Test Report* is created based on the test results and test analysis. A subset of or the entire test cases will be re-executed to validate the fixes that a Developer develops to remedy the confirmed vulnerabilities and exclude any regression.

In such a model-driven penetration test campaign, a Penetration Tester does not directly work out concrete test cases. Instead, based on the test campaign model, a

Penetration Tester systematically collects test data from the development domain and the web security domain, and then populates the model step by step with the test data to produce a set of test artifacts in a data-driven approach.

4.8. Knowledge-Based PenTest Workbench

Based on the test campaign model and the grey-box test architecture, a knowledge-based PenTest workbench can be developed to support the entire test process from test analysis to test execution. Figure 16 shows the core functionalities such a PenTest workbench should support. The purpose of the PenTest workbench is to support these functionalities by leveraging the artifacts represented in the penetration test campaign model as well as the knowledge in the web security knowledgebase and to use third party tools to enable tool-based execution of the penetration test campaign. With the core functionalities integrated into one test environment, output of one functionality can be used seamlessly as input to another.

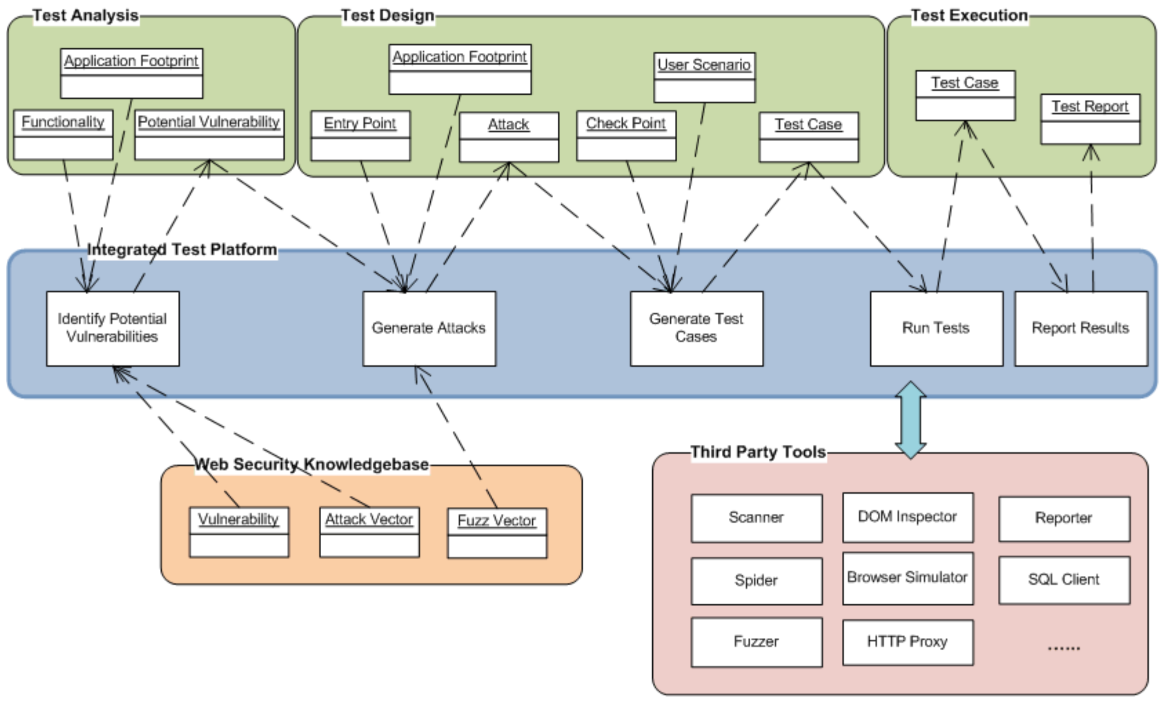


Figure 16 A Knowledge-Based PenTest Workbench

Below is an explanation of each of the core functionalities. In Chapter 5, we will discuss how the functionalities were implemented in our case studies.

- **Identify Potential Vulnerabilities**: at the test analysis phase, Application Footprint and application Functionalities are mapped to Vulnerabilities and Attack Vectors in order to generate a list of Potential Vulnerabilities.
- **Generate Attacks**: at the test design phase, the Potential Vulnerabilities, Entry Points, Application Footprint, and Fuzz Vectors are used to generate Attacks.
- **Generate Test Cases**: at the test design phase, the Attacks, Check Points, and User Scenarios are used to generate a set of Test Cases.
- **Run Tests**: at the test execution phase, the Test Cases are loaded and run, with the test results captured and saved.
- **Report Results**: at the test execution phase, a Test Report is generated based on the test results.

Chapter 5. Test Campaign Case Studies

The test framework proposed in Chapter 3 was evaluated and refined by conducting three test campaigns with respect to the penetration test methodology, the grey-box test architecture, the web security knowledgebase, the test campaign model, and the PenTest workbench. The first test campaign was run against a reference test web application. The second and third test campaigns were run against real web applications.

In section 5.1, we give an overview of the three test campaigns. In section 5.2, the implementation of the test framework prototype that was applied to the test campaigns is explained. Section 5.3 describes the test campaign for WebGoat. Section 5.4 describes the test campaign for AEMS. Section 5.5 describes the test campaign for PAL-IS.

5.1. Overview of Test Campaigns

5.1.1 WebGoat Test Campaign

The first test campaign was run with WebGoat (OWASP WebGoat Project, 2011), a deliberately insecure J2EE web application developed and maintained by the Open Web Application Security Project (OWASP). The purposes of this test campaign were to level set the web security knowledgebase and verify the test campaign model and the PenTest workbench. However, since WebGoat is an existing, completed reference web application, the test campaign was an isolated test activity and there was no interaction with developers.

5.1.2 AEMS Test Campaign

The second test campaign was run with AEMS – Adverse Event Management System (Behnam, Amyot, Forster, Peyton, & Shamsaei, May, 2009). The application was developed on the Microsoft .Net platform. The test campaign started in the middle of the development of AEMS during the development of a beta version.

The Development Team

AEMS was developed by a team at the University of Ottawa. The developers on the AEMS team were aware of the common web application vulnerabilities that might exist in the application. They had some knowledge of the .Net security and relied on the .Net built-in security mechanisms and ibatis (mybatis Home Page, 2011), a third-party persistence framework, to protect the application from the attacks exploiting the vulnerabilities. But they did not exactly understand how the security mechanisms work so they were not sure if the vulnerabilities had been remediated or not. The developers were aware of the Microsoft Security Development Lifecycle (the MS SDL) principles and tools, and stated that they followed the MS SDL in the application development. However, from our discussions and observations, their adherence to MS SDL seemed ad hoc at best. They planned to engage an external security expert to assess the security posture of AEMS and address any security issues at the verification phase.

The Security Development Practices

The development team more or less followed the MS SDL process in AEMS development as follows:

- A primary developer took an undergraduate security course for the education in web application security fundamentals.
- The development team performed a threat analysis to understand the potential vulnerabilities might exist in AEMS, which include Stored Cross-Site Scripting and Standard SQL Injection.
- The development team analyzed the potential vulnerabilities. Based on their knowledge, they thought the build-in security mechanism in ASP.Net could protect AEMS from Stored Cross-Site Scripting attacks, and ibatis could prevent AEMS from Standard SQL Injection attacks. The development team did not implement any application-specific security measures for the potential vulnerabilities as they left it as an effort at the verification phase.
- The development team planned to engage an external security expert to perform penetration testing at the verification phase to assess the security posture of AEMS, and to address any security issues then as they arose.
- The Penetration Test Framework was introduced into the AEMS project during the development of the beta version of AEMS. There were two iterations of penetration testing that tested a representative subset of AEMS functionalities. One false negative was fixed, and one true negative was identified but the team chose not to fix it at that time.

The Refinement of the Test Framework

Based on the experience with the AEMS test campaign, the test framework was refined significantly (section 5.4.6 discusses the refinement in more detail). The refined framework was then applied to the PAL-IS test campaign.

5.1.3 PAL-IS Test Campaign

The third test campaign was run with PAL-IS – Palliative Care Information System. The application was developed on the Microsoft .Net platform. The test campaign was fully integrated into the security-oriented software development life cycle for the application.

The Development Team

PAL-IS was developed by a different team at the university of Ottawa. The developers on the PAL-IS team were aware of the common web application vulnerabilities that might exist in the application. They had some knowledge of the .Net security and knew there were some built-in security mechanisms in .Net, but they did not know how the security mechanisms worked and therefore did not have any idea which vulnerabilities had been remediated and which had not. The developers were aware of the Microsoft Security Development Lifecycle (the MS SDL) principles and tools, and they would follow the MS SDL in the application development (although, based on our observations, it appeared to be done in an ad hoc manner at best). The AEMS team had more experience in web application security than the PAL-IS team when the both started their projects, e.g. the AEMS development team had knowledge of the typical XSS and Standard SQL Injection vulnerabilities in web applications and the defense mechanism in ASP.Net and ibaits.

The Security Development Practices

We worked with the development team by following the proposed test methodology and leveraging the prototyped PenTest workbench:

- At the analysis phase, the penetration tester identified a list of potential vulnerabilities that might exist in PAL-IS, and sent it to the development team so that they were aware of the potential threats and could design appropriate countermeasures.

- At the design phase, the development team analyzed the potential vulnerabilities. They thought the build-in security mechanism in .Net would provide protections for PAL-IS against Stored Cross-Site Scripting and Standard SQL Injection attacks. They finished the PAL-IS design without considering application specific security measures against the two types of vulnerabilities. In parallel, the penetration tester developed test cases based on a representative subset of the functionality in PAL-IS.
- At the implementation phase, there were three rounds of penetration testing. After the first round of testing, the penetration tester worked with the development team to analyze the test results. An additional check point in the application logs was added to eliminate suspicious false negatives in the second round. Five XSS test cases passed as the result of the more comprehensive check points, and five true negatives were identified this time. The Standard SQL Injection vulnerability was then remedied, and all the four SQL Injection test cases passed in the third round of testing. There was one true negative XSS test case that the team chose not to address at that time.

5.1.4 The Roles of the Thesis Researcher in the Case Studies

The thesis researcher introduced, coordinated, and iteratively improved the framework during the case studies (especially after the AEMS case study). The framework was introduced to both development teams. The thesis researcher showed them how a tester uses the framework. The development teams agreed to adopt the framework instead of following an approach where they brought in an expert at the end of development cycle for the application security assessment (with AEMS they were already in the middle of the development cycle). The thesis researcher coordinated with the person responsible for testing and played the major role of penetration tester using the framework.

The thesis researcher got informal feedback from the development teams (their tester, their developers and the development manager) for the case studies on what it was like to try out the framework. In particular, these feedbacks were the basis for many of the improvements that were made after the AEMS case study and before the PALIS case study. The improvements resulted in the test artifacts/documents that were used in the PALIS case study.

The thesis researcher does not feel himself in anyway to be a security expert. In the case studies, the researcher consulted with websites and textbooks, captured web security knowledge as best he could, and then built the sample web security knowledgebase. The thesis researcher received feedback from Prof. Guy-Vincent Jourdan and Prof. Anil Somayaji in the role of security experts on the web security knowledgebase during the thesis proposal review after the AEMS case study was completed and before the PALIS case study was done.

5.2. Test Framework Prototype

A prototype version of the proposed test framework was implemented and was used for all three test campaigns. The test framework prototype includes a Java-based PenTest workbench that facilitates the test tasks such as potential vulnerability identification, test case generation, test execution, and test reporting, along with a test campaign model and a sample web security knowledgebase. The test framework was significantly refined based on the experience with the AEMS test campaign (section 5.4.6 discusses the refinement in more detail).

5.2.1 Test Artifacts and Documents

Below are the test artifacts in a test campaign that are created at each phase of the software development life cycle:

- **Test Analysis Document:** to document the test artifacts and relevant web security artifacts created at the test analysis phase, including *Application Footprint*, *Potential Vulnerability and Attack Vector*, and to list the web security and development artifacts utilized at this phase, including *Functionality*, *Use Case* and *Functional Specification*.
- **Test Design Document**: to document the test artifacts created at the test design phase, including *Entry Point*, *Attack*, *Check Point*, and *Test Case*, and to list the development artifacts utilized at this phase, including *Web API*, *System Architecture*, and *User Scenario*.
- **Test Execution Document:** to document the test artifacts created at the test execution phase, including *Test Result* and *Test Report*.

5.2.2 Test Campaign Model

The general test campaign model showed in Figure 15 was implemented in the test campaigns as specified in Figure 17. The artifacts fall into three categories: test campaign artifacts, development artifacts, and web security knowledgebase.

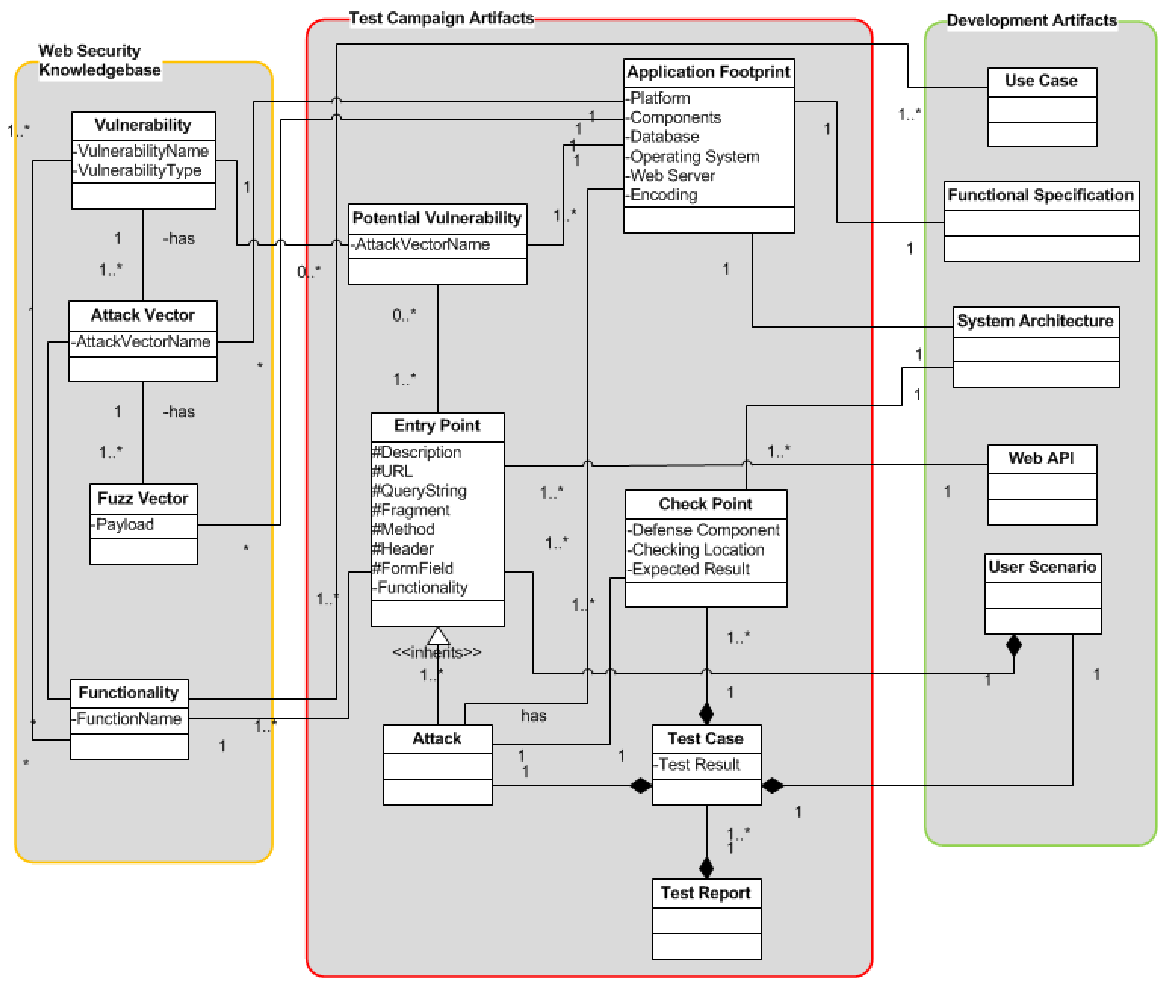


Figure 17 Test Campaign Model

In the WebGoat test campaign and the AEMS test campaign, Entry Point was defined as an HTTP request, and Check Point was defined as an HTTP response. In the PAL-IS test campaign, Entry Point was defined as user inputs on web pages in browser, and Check Point was defined as content in web pages rendered in browser and an entry in the application logs at server-side. Check Point is linked to Application Footprint by specifying in a check point the component that is to trap an attack.

Attack is a special entry point in that one of its fields is manipulated with a fuzz vector payload. An attack has one or more check points.

Use Case, Functional Specification, Web API, and System Architecture were not implemented as a table in database. They were defined in MS Word documents that were processed manually. User Scenarios, however, were entered and stored in the database.

5.2.3 Knowledge-Based PenTest Workbench Implementation

A prototype knowledge-based PenTest workbench, as discussed in Chapter 4, was implemented for the three test campaigns. It uses the following open source technologies:

- J2SE 1.6 (Java SE 6, 2011): it is used to build up the PenTest workbench
- HttpClient 4.0 (Apache HTTPClient Home Page, 2011): a third-party component used to process http requests and http responses
- MySQL Server 5.1.34 (MySQL Home Page, 2011): it is used to permanently store test artifacts and the sample web security knowledgebase
- Eclipse 3.4.2 (Eclipse Home Page, 2011): IDE used to develop the PenTest workbench
- SQuirreL 3.0.1 (SQuirreL SQL Client Home Page, 2011): a SQL client used to maintain the web security knowledgebase and the test campaign database
- WebScarab: an HTTP proxy used to assist in test design and test execution

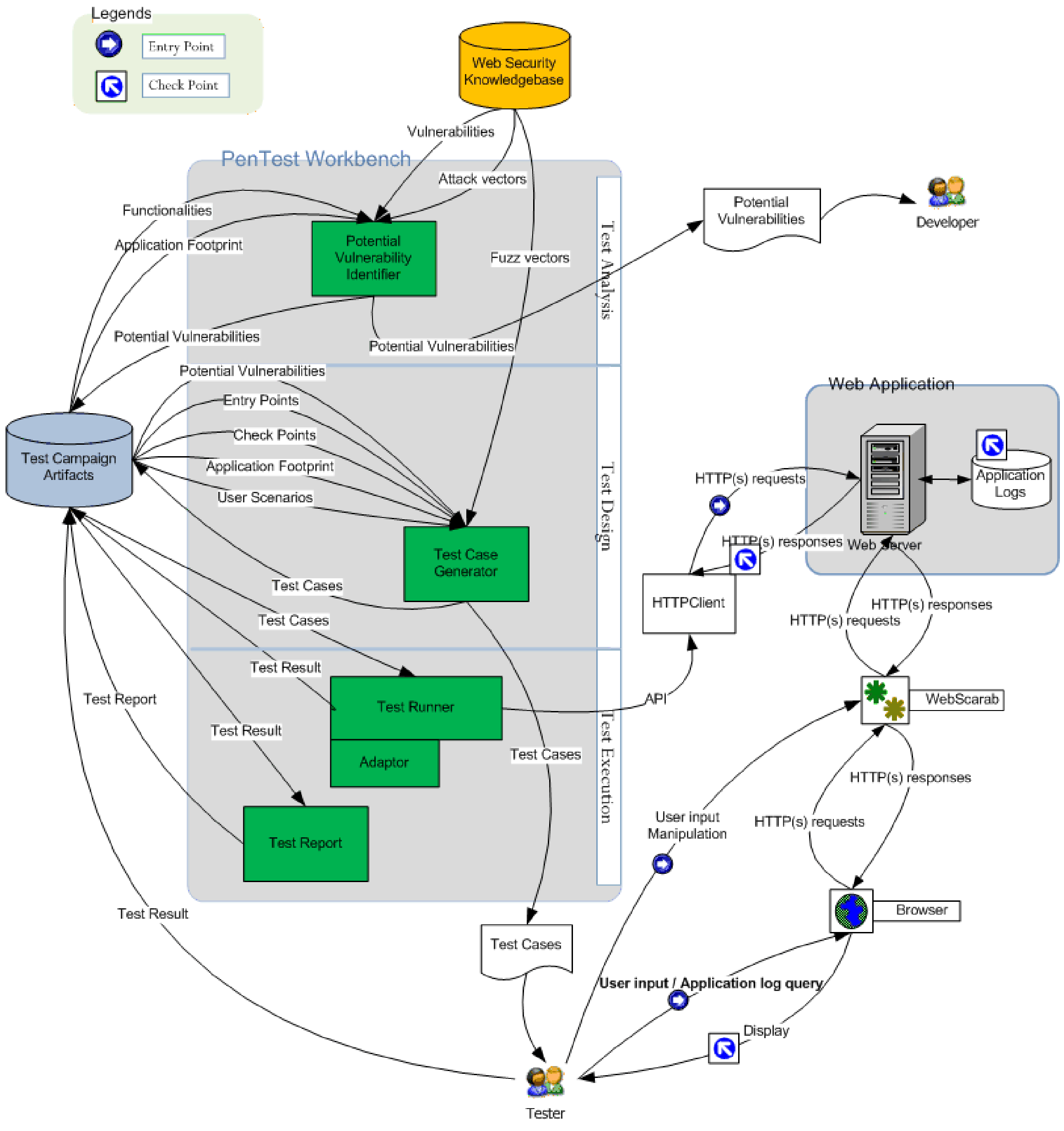


Figure 18 Test Architecture and the Knowledge-Based PenTest Workbench

Figure 18 shows the test architecture and the PenTest workbench that was used in the test campaigns. The PenTest workbench consists of five major tools:

- Potential Vulnerability Identifier

Potential Vulnerability Identifier is a generic tool that generates all potential vulnerabilities and feasible attack vectors for a given web application based on its functionalities, the application footprint, and the web security knowledgebase. It saves the identified potential vulnerabilities and attack vectors to the test campaign database, and can print out the list of the identified vulnerabilities and attack vectors so that it can be sent to the development team.

- Test Case Generator

Test Case Generator is a generic tool. For each entry point, based on the identified potential vulnerabilities and attack vectors, and the application footprint, it generates test attacks by applying relevant and feasible fuzz vectors. Then for each attack and its check point, Test Case Generator picks up a feasible user scenario based on a pre-defined algorithm, e.g. shortest paths, to generate a test case. Test Case Generator saves the generated test cases to the test campaign database, and can print out the test cases so that a tester can read and follow the test steps to execute them manually.

- Test Runner

Test Runner is a generic test tool that loads and executes test cases as sequences of HTTP(s) requests, and verifies test results as HTTP(s) responses against expected ones. The Test Runner uses an application-specific Test Runner Adaptor to handle application specific settings and processing.

- Test Runner Adaptor

Test Runner Adaptor is an HTTP(s) messaging based utility that handles application specific setting or processing such as creating SSL connections, generating dynamic query strings, etc. on behalf of the Test Runner. An application specific adaptor was written for each of WebGoat and AEMS test campaigns. However, because of the complexity of .Net generated HTTP requests in PAL-IS, User Inputs in browser, rather than HTTP requests, were used to drive tests so that the HTTP Test Runner was not needed.

- Test Report

Test Report is a generic utility for documenting and summarizing the test results from a test execution in a test campaign.

- Database

The sample web security knowledgebase was implemented using MySQL. Each web security object (including Vulnerability, Attack Vector, Fuzz Vector, and Functionality) defined in Figure 17 is represented as a table. The implementation from the conceptual model is straightforward. Primary keys and foreign keys were created when necessary. In addition, a set of SQL scripts was developed to load sample vulnerabilities, attack vectors and fuzz vectors (discussed in detail in section 5.2.4). For those special characters in the sample fuzz vectors, backsplash escape is used in the SQL scripts. There are 5 rows in the Vulnerability table, 9 rows in the Attack Vector table, and 26 rows in the Fuzz Vector table.

Similarly, the test campaign and development artifacts, except Use Case, Functional Specification, System Artifact and WEB API, were implemented in the same database. We did not capture the four development artifacts in the database because they are not the focus of the thesis research, e.g. they do not have significant impact on the design of the proposed test framework in this thesis.

5.2.4 Web Security Knowledgebase

The sample web security knowledgebase used in the three test campaigns was gathered from several web security knowledgebases available publically on the Web, including OWASP TOP 10 Web Application Vulnerabilities (OWASP TOP 10, 2007), OWASP Testing Guide (OWASP Testing Guide, 2008), XSS cheat sheet (Hansen, 2011), and SQL Injection cheat sheet (Daw, SQL Injection Cheat Sheet, 2009). The five most prevalent web application vulnerabilities, namely Reflected XSS, Stored XSS, DOM Based XSS, Standard SQL Injection, and Cross Site Request Forgery (CSRF), and their related attack vectors and fuzz vectors, were gathered and represented in the web security knowledgebase. Figure 19 shows a sample vulnerability entry. Figure 20 shows a sample attack vector entry. Figure 21 shows a sample fuzz vector entry.

```
Vulnerability: Reflected XSS
Vulnerability Type: Cross-Site Scripting
Functionalities: Search; Personalize a Welcome Page with User Name; Cause an
error message response with user inputs embedded
```

Figure 19 A Sample XSS Vulnerability

```
Vulnerability: Reflected XSS
Attack Vector: Manipulate the input field whose value is to be searched
Application Footprints:
  Platform: *
  Components: *
  Database: *
  Operating System: *
  Web Server: *
  Character Encodings: *
```

Figure 20 A Sample Attack Vector for XSS

```
Vulnerability: Reflected XSS
Attack Vector: Manipulate the input field whose value is to be searched
Payload: <script>alert("This is a sample XSS attack");</script>
Application Footprints:
  Platform: *
  Components: *
  Database: *
  Operating System: *
  Web Server: *
  Character Encodings: *
```

Figure 21 A Sample Fuzz Vector for XSS

In the sample web security knowledgebase, five functionalities were categorized and populated. They are the samples of most common functionalities that one sees in web applications, as discussed in section 2.1.3. The following case studies focused on the sample set of functionalities.

Login: Login is an authentication procedure used to get access to an application, e.g. Form-based username/password Login. It may be vulnerable to Standard SQL Injection attacks.

Search: Search is a common function in a web application that finds relevant information based a set of criteria, e.g. Search for a user, Search Patient etc. It may be vulnerable to Standard SQL Injection attacks and Reflected Cross-Site Scripting attacks.

Form Data Collection: HTML form is often used in web application for general data entry. It may be vulnerable to Stored Cross-Site Scripting attacks.

User Profile Management (Edit/View): It is a common function in a web application to edit and/or review users' profile. It may be vulnerable to Stored Cross-Site Scripting attacks.

Load Attachments: Some web applications may be able to load attachments, e.g. pdf files, and then save them on server that can be accessed by users later on. It may be vulnerable to Stored Cross-Site Scripting attacks.

The web security knowledgebase used in the case studies was implemented in a relational database.

5.3. WebGoat Test Campaign

5.3.1 Introduction

WebGoat is a deliberately insecure J2EE web application for the purpose of web application security training and a benchmark for security test tools evaluation (OWASP WebGoat Project, 2011). WebGoat runs over plaintext HTTP. It uses HTTP basic authentication. The WebGoat system studied in this case study is version 5.2 and it was deployed in a Lab environment.

The primary purposes of this case study were to level set the web security knowledgebase, and verify the test campaign model and the PenTest workbench. WebGoat is an existing, completed reference web application. We were not involved with its development process, nor was there any interaction with its developers. All the test artifacts in the WebGoat test campaign were created by various indirect ways such as inspecting the WebGoat installation, searching the relevant information on the Web, checking WebGoat inline help, etc., instead of being based on the development artifacts as specified in the test methodology in section 4.4. The test campaign was in effect black box testing. Therefore we can only evaluate the test campaign model, the PenTest workbench, and the sample web security knowledgebase in this case study.

5.3.2 Test Requirement Analysis

WebGoat *Application Footprint* was gathered by inspecting the WebGoat installation and searching the relevant information on the Web. Figure 22 is the application footprint of WebGoat v5.2.

```
Platform: J2EE 1.4, JRE 1.6.0_01
Components: -
Database: Derby 10.2.1.6
Operating System: Windows XP
Web Server: Apache Tomcat 5.5
Encoding: UTF8
```

Figure 22 WebGoat Application Footprint

WebGoat v5.2 has over 30 lessons that illustrate various web application vulnerabilities. The three most common vulnerabilities, *Reflected XSS*, *Stored XSS*, and *Standard SQL Injection*, were selected and tested against WebGoat in this case study, as shown in Appendix A1. Accordingly, four scenarios were selected that have one of the vulnerabilities, and then the four scenarios were entered into the database manually. Figure 23 shows a scenario that has an entry point containing a Reflected XSS vulnerability. All the test scenarios used in the WebGoat test campaign are listed in Appendix A2.

```
User Scenario 1: Phishing with XSS
Step1: Launch WebGoat Home Page
Description: In Browser, enter URL "http://127.0.0.1:80/WebGoat/attack" to open
login page.

Step2: Log on using HTTP Basic Authentication
Description: Enter "guest" as username, "guest" as password, and click on "Log
In" button to log on WebGoat.

Step3: Launch WebGoat Course Page
Description: Click on "Start WebGoat" button to launch WebGoat course page.

Step4: Navigate to the "Phishing with XSS" page
Description: Click on "Cross-Site Scripting (XSS)" navigation menu at the left
of the web page to expand it, and then click on "Phishing with XSS" link to
launch the page.

Step5: Search Employee
Description: Enter "Tom" in the search text field, and then click on "Search"
button.

Step6: Entry Point: Logout
Description: Click on "Logout" link to log out WebGoat.
```

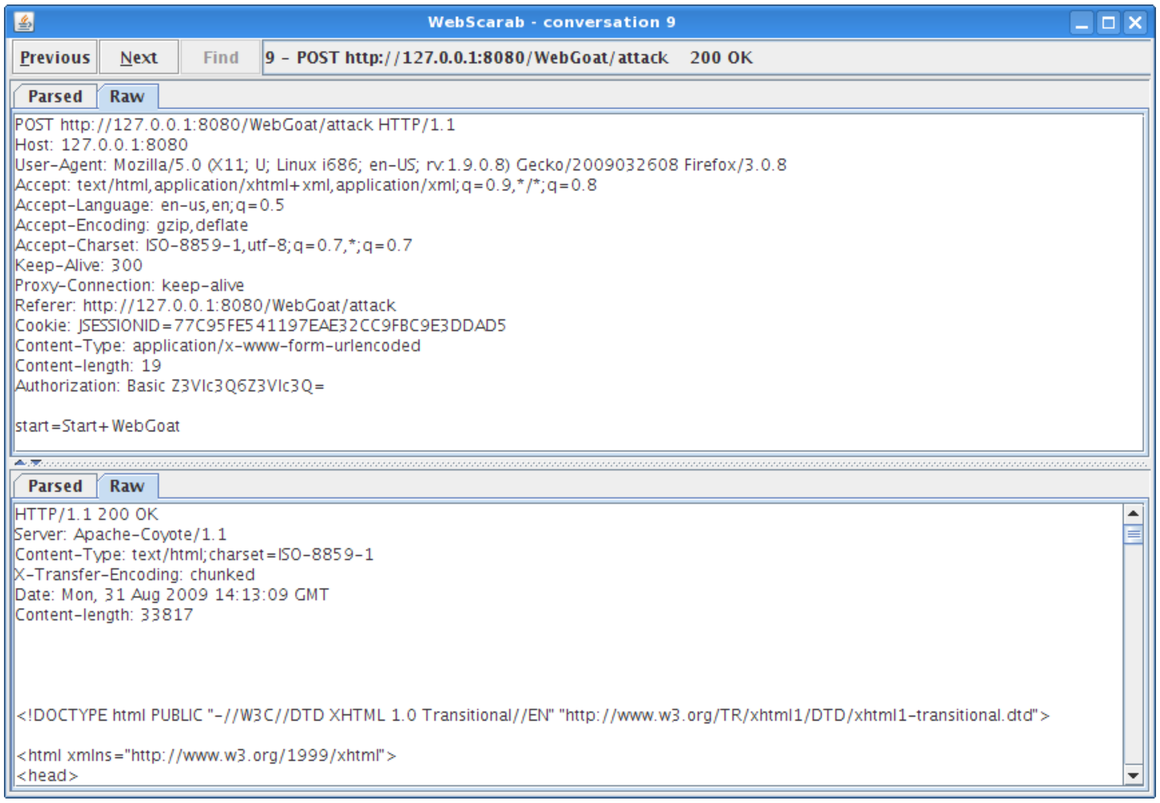
Figure 23 A Sample User Scenario in the WebGoat Test Campaign

5.3.3 Test Case Design

The test design followed the four major steps below:

Identify Entry Points

In this test campaign, an *Entry Point* is an HTTP request. Based on the scenarios identified above, the entry points in the scenarios were identified manually using a HTTP proxy called WebScarab. Figure 24 shows how an HTTP request can be captured and intercepted using WebScarab.


WebScarab - conversation 9
Previous
Next
Find
9 - POST http://127.0.0.1:8080/WebGoat/attack 200 OK
Parsed
Raw

```
POST http://127.0.0.1:8080/WebGoat/attack HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.8) Gecko/2009032608 Firefox/3.0.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
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
Proxy-Connection: keep-alive
Referer: http://127.0.0.1:8080/WebGoat/attack
Cookie: JSESSIONID=77C95FE541197EAE32CC9FBC9E3DDAD5
Content-Type: application/x-www-form-urlencoded
Content-length: 19
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=

start=Start+WebGoat
```

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=ISO-8859-1
X-Transfer-Encoding: chunked
Date: Mon, 31 Aug 2009 14:13:09 GMT
Content-length: 33817



<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

Figure 24 Use WebScarab to identify entry point

Functionality for each entry point was also identified at this step. The WebGoat web application itself was used to identify the functionalities. Figure 25 is a sample entry point. All the entry points used in this case study are listed in Appendix A2.

```
Entry point: Launch WebGoat Home Page (ID: ep001)
GET http://127.0.0.1:80/WebGoat/attack
Functionality: Navigation
```

Figure 25 A Sample Entry Point in the WebGoat Test Campaign

Generate Attacks

For each entry point, the potential vulnerabilities and the feasible attack vectors were identified, based on the functionality of the entry point, using the tool "Potential Vulnerability Identifier" [1] (shown in Figure 18). We compared the list of identified vulnerabilities with the WebGoat documentation to ensure that we had proper coverage. Then based on the identified potential vulnerabilities, we identified the parameters to be manipulated to create attacks. *Attacks* were then generated using a utility in the tool "Test Case Generator" which applies feasible fuzz vector payloads to the manipulated parameters of the entry points with the consideration of the application footprint. The generated test attacks were saved as interim results in the database. Figure 26 shows two attacks against entry point "Search Employee".

```
Entry point: Search Employee

Attack: 1
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field: Username=<script>alert('XSS');</script>

Attack: 2
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field: Username=>"><script>alert('XSS')</script>&
```

Figure 26 Sample Attacks in the WebGoat Test Campaign

[1] Figure 18 shows the latest test architecture that was refined after the AEMS test campaign. In this case study, "Potential Vulnerability Identifier" was a utility used in the test design phase.

Identify Check Points

A *Check Point* in the WebGoat test campaign is an HTTP response. One check point was identified for each attack. The course instruction in WebGoat, which explains how an attack is constructed and how it was handled in WebGoat, was used to identify the check points. Figure 27 shows a sample attack with a check point.

```
Entry point: Search Employee

Attack: 1
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field: Username=<script>alert('XSS');</script>

Defense Mechanism: None
Check Point: "Search Employee" Page
   Expected result -> the response page is displayed with a popup window with
"XSS" message.
```

Figure 27 A Sample Check Point in the WebGoat Test Campaign

Generate Test Cases

Based on the selected scenarios, for each attack, a *Test Case* was generated using the tool "Test Case Generator" (shown in Figure 18). All the generated test cases are listed in Appendix A2.

5.3.4 Test Execution

The generated test cases were loaded and executed automatically using the tool "Test Runner" (shown in Figure 18), and the *Test Results* were saved in the database. Each test step was sent out by "Test Runner" as an HTTP request in sequence. If an HTTP response was a check point, "Test Runner" checked the response against the specified expected result to determine if the attack succeeded or not. "Test Runner" was designed as a generic tool to any test campaign. It called the application specific utility "Test Runner Adaptor" to handle those application specific tasks. In the WebGoat test campaign, basic authentication and dynamic URL parsing were handled by the "Test Runner Adaptor".

When the test execution was completed, the tool "Test Report" was run to generate and print out the *Test Report* based on the test results. The test report was documented in the test execution document (Appendix A3).

5.3.5 Test Result Summary

Table 3 shows the WebGoat test campaign summary.

Table 3 The WebGoat Test Campaign Summary

Web Security Knowledgebase	
Vulnerability	5
Attack Vector	9
Fuzz Vector	26
Test Effort	
Entry Points	13
Potential Vulnerable Entry Points	4
Test Cases	13
Test Result	
3 types of vulnerabilities were identified on 4 entry points correctly	

Table 4 is the test result summary by comparing the identified vulnerabilities to its intended design. It shows that all the planted vulnerabilities were identified successfully, and no identified vulnerability was a false negative.

Table 4 The WebGoat Test Result Summary

Entry Point	Intended Vulnerability	Identified Vulnerability	Comments
ep001	None	None	True Positive
ep002	None	None	True Positive
ep003	None	None	True Positive
ep004	None	None	True Positive
ep005	Reflected XSS	Reflected XSS	True Negative
ep006	None	None	True Positive
ep007	None	None	True Positive
ep08	Stored XSS	Stored XSS	True Negative
ep009	None	None	True Positive
ep010	None	None	True Positive
ep011	Numeric SQL Injection	Numeric SQL Injection	True Negative
ep012	None	None	True Positive
ep013	String SQL Injection	String SQL Injection	True Negative

5.4. AEMS Test Campaign

5.4.1 Introduction

AEMS is a .Net based web application that was developed at the University of Ottawa to manage adverse events at the Ottawa Hospital. AEMS is configured to run over HTTPS. It uses form-based authentication. An alpha version of AEMS had been implemented and deployed (Behnam, Amyot, Forster, Peyton, & Shamsaei, May, 2009). The application studied in this case study was a beta version of AEMS hosted in a development environment, where it was under active development.

The primary purpose of this case study is to evaluate the test framework by applying it to a real web application. The penetration test campaign was introduced during the development of a beta version of AEMS, therefore we could not fully validate the integrated penetration test process. In addition, not all of the required development artifacts were available that could be utilized directly in the penetration test development according to what is described in the test methodology in section 4.4. Instead, we met with the development team to informally elicit the relevant information (if it was missing), and developed the test artifacts based on the gathered information.

5.4.2 Test Requirement Analysis

The development artifacts, including *Functional Specification* and *Use Case*, were acquired informally at a system walkthrough meeting held by the development team. Based on the information gathered at the meeting, the *Application Footprint* for AEMS was identified (Figure 28). A subset of *User Scenarios* that would be studied in this test campaign was documented. Figure 29 shows a sample user scenario. All the user scenarios used in this case study are listed in Appendix B2.

```
Platform: .NET Framework (.NET ASP) 3.5 SP1
Components: -
Database: SQL Server 2005
Operating System: Windows 2008 Server
Web Server: IIS 7.0
Encoding: *
```

Figure 28 AEMS Application Footprint

```
User Scenario 1: Login and then Logout
Step1: Entry Point: Launch Login Page (ID: ep001)
Description: In Browser, enter URL
"HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f" to open login
page

Step2: Entry Point: Log On (ID: ep002)
Description: On the login page, input "pxiong" for field "username", input
"123456" for field "password", and click on "Log In" button to log in AEMS as
user "pxiong"

Step3: Entry Point: Logout (ID: ep003)
Description: Click on "logout" button to log out AEMS
```

Figure 29 A Sample User Scenario in the AEMS Test Campaign

5.4.3 Test Case Design

The test design followed the four major steps below:

Identify Entry Points

In the AEMS test campaign, an *Entry Point* is an HTTP request. Based on the scenarios identified above, the entry points in the scenarios were identified by walking through the application during the meeting with the development team, and were confirmed by using WebScarab to capture and intercept the HTTP requests. *Functionality* for each entry point was also identified at this step. Figure 30 is a sample entry point. All the entry points used in this case study are listed in Appendix B2.

```
Entry point: Launch Login Page (ID: ep001)
GET HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Functionality: Navigation
```

Figure 30 A Sample Entry Point in the AEMS Test Campaign

Generate Attacks

For each entry point, based on its functionality and the application footprint, the *Potential Vulnerability* and the feasible attack vectors were identified using the tool "Potential Vulnerability Identifier"[2]:

Input: Entry Point, Application Footprint, Vulnerability, Attack Vector

Manual Action: None

Tool: Potential Vulnerability Identifier

Output: Potential Vulnerability, feasible Attack Vector

Then based on the identified potential vulnerabilities and the feasible attack vectors, the penetration tester manually identified the parameter(s) in the entry point to be manipulated to create attacks. *Attack* was then generated using a utility in the tool "Test

[2] Figure 18 shows the latest test architecture that was refined during the PAL-IS test campaign. In this case study, "Potential Vulnerability Identifier" was a tool used in the test design phase.

Case Generator" which applies feasible fuzz vector payloads to the manipulated parameters of the entry points based on the application footprint. The generated attacks were saved as interim results in the database:

Input: Entry Point, Potential Vulnerability, feasible Attack Vector, Fuzz Vector

Manual Action: Identify parameter(s) to be manipulate

Tool: Test Case Generator

Output: Attack

Figure 31 shows two attacks against entry point "Log On".

```
Entry point: Log On (ID: ep002)

Attack: 1
Attack Description: Standard SQL Injection attack
POST HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Manipulated Form Field: password=1 or 1=1--

Attack: 2
Attack Description: Standard SQL Injection attack
POST HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Manipulated Form Field: password=a' or 'a'='a
```

Figure 31 Sample Attacks in the AEMS Test Campaign

Identify Check Points

A *Check Point* in the AEMS test campaign is an HTTP response that is rendered in a browser as a web page. The check points were identified by walking through the application. One check point was identified for each attack. Figure 32 shows a sample check point.

```
Entry point: Log On (ID: ep002)

Attack: 1
Attack Description: Standard SQL Injection attack
POST HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Manipulated Form Field: password=1 or 1=1--

Defense Mechanism: ibatis component
Check Point: "Login" Page
   Expected result -> the Login web page is displayed with error message: Your
login attempt was not successful. Please try again.
```

Figure 32 A Sample Check Point in the AEMS Test Campaign

Generate Test Cases

Based on the selected scenarios in section 5.4.2, for each attack, a *Test Case* was generated using the tool "Test Case Generator" (shown in Figure 18):

Input: Attack, Check Point, User Scenario

Manual Action: None

Tool: Test Case Generator

Output: Test Case

All the generated test cases are listed in Appendix B2.

5.4.4 Test Execution

The generated test cases were loaded and executed automatically using the tool "Test Runner" (shown in Figure 18), and the *Test Results* were saved in the database. Each test step was sent out by "Test Runner" as an HTTP request in sequence. If an HTTP response was a check point, "Test Runner" checked the response against the specified expected result to determine if the attack succeeded or not. An application specific "Test Runner Adaptor" was developed for the AEMS test campaign, including HTTPS connection and form based authentication.

First Round of Testing

Two vulnerabilities were identified in the first round of testing: Standard SQL Injection on the entry point ep006 and Stored XSS on the entry point ep004.

The test result was sent to the development team. The development team performed code reviews to analyze the Standard SQL Injection vulnerability. A piece of source code (Figure 33) that was relevant to the defect shows that the Standard SQL Injection vulnerability was not a real vulnerability. It was a false negative. It was actually a functional defect in the code of an application specific input validation & filtering module – when a search text contains any non-alphabetic character, instead of rejecting the input as an invalid search text (since it is reasonably assumed that nobody's name contains non-alphabetic character), it converts the search text to "" which is in turn treated as "search all patients". As a result, the test returned all patients in the database as the search result, which looked like a successful SQL Injection attack.

```
if (searchID == lastname || searchID == firstname )
   {
       if (IsAlpha(textSearched))
       {
           return textSearched;
       }
       else {
           textSearched = "";
       }
   }
```

Figure 33 "Validator" Code Snippet in AEMS

For the Stored XSS vulnerability, the penetration tester did some research on the Web. The test failure is due to a known issue in .Net 3.5 SP1 (XSS vulnerabilty in ASP.Net, 2005). Although in general, .Net 3.5 SP1 framework provides a built-in security mechanism against Stored XSS attacks, it fails to trap a special fuzz vector that uses an obscure character encoding that was used in the test. This is a known defect in

.Net, that Microsoft has no plans to fix as it was considered a very rare case and does not have serious security impact, so the development team felt that it could be ignored as well, at least for the Beta release. This security issue was analyzed more extensively in the PAL-IS test campaign. Section 5.5.4 has more detailed discussion on this.

Second Round of Testing (Regression Testing)

When the functional defect related to the test of Standard SQL Injection was fixed, the same set of test cases was executed. In this case, since the defect fix did not change the entry point, there was no need to update the test cases. The second round of test was simply to re-run the existing test cases automatically using the tool "Test Runner". The Standard SQL Injection false negative was eliminated this time.

When the test execution was completed, the tool "Test Report" was run to generate and print out the *Test Report* based on the test results. The test report was documented in the test execution document (Appendix B3).

5.4.5 Test Result Summary

Table 5 shows the AEMS test campaign summary.

Table 5 The AEMS Test Campaign Summary

Web Security Knowledgebase	
Vulnerability	5
Attack Vector	9
Fuzz Vector	26
Test Effort	
Entry Points	6
Potential Vulnerable Entry Points	3
Test Cases	10
Test Result	
First round	2 test cases failed. 2 types of vulnerabilities were identified on 2 entry points
Second round	1 test case failed. 1 type of vulnerability (known MS XSS) was identified on 1 entry point

5.4.6 The Refinement of the Test Framework

With the experience in the AEMS test campaign, the test framework was refined significantly:

- The test methodology was fully integrated into the security-oriented development life cycle – from the analysis phase to the application maintenance phase.
- In each test phase, the utilized development artifacts and the created test artifacts were precisely defined. A set of templates for documenting the test artifacts were also well defined.
- In each test phase, the communication between developer and penetration tester were well defined.
- The test artifact *Check Point* was precisely defined to represent various types of check points on the both client-side and server-side. In addition, the systematic approach to identify check points was also defined. By following the process and with the collaboration between developer and penetration tester, every test case is supposed to have a check point that contains adequate information to determine if an attack has been trapped or not.

5.5. PAL-IS Test Campaign

5.5.1 Introduction

PAL-IS is a .Net based web application that was under active development at the University of Ottawa for the palliative pain and symptom management consultation service. PAL-IS has been developed iteratively as a series of prototypes. A beta release of the application is now deployed as a pilot project at the Élisabeth Bruyère Hospital in Ottawa.

The primary purpose of this case study is to evaluate the entire test framework that was refined significantly after the AEMS test campaign. The penetration testing effort was started at the beginning of the development of PAL-IS when the very first PAL-IS prototype was implemented and deployed in a test environment. The penetration test campaign has been integrated with the development lifecycle from the initial analysis to the prototype design to the alpha testing and then the beta release of the application.

5.5.2 Test Requirement Analysis

At the system analysis phase, development artifacts – *Use Cases* and *Functional Specifications*, are utilized to create test requirement artifact – *Application Footprint* and web security artifact – *Functionality*. In this test campaign, specifically, the *PAL-IS Project Specification document* and *PAL-IS Use Cases (for Demo) document*, were used to identify *PAL-IS Application Footprint* (Figure 34) and the *PAL-IS Functionalities* (Figure 35).

```
Platform: .NET Framework (ASP.NET) 3.5 SP1
Components: -
Database: SQL Server 2005
Operating System: Windows 2008 Server
Web Server: IIS 7.0
Encoding: *
```

Figure 34 PAL-IS Application Footprint

```
1. Login: User login
2. Search Patient: Search a patient or patients
3. Form Data Collection: Various form-based data view and edit
4. User Profile Management: User profile view and edit
5. Load Attachments: Load and view attachments
```

Figure 35 PAL-IS Functionalities

Based on the security knowledgebase, the functionalities and the application footprint, a set of *Potential Vulnerability* and related attack vectors against PAL-IS were generated using the tool "Potential Vulnerability Identifier" (shown in Figure 18):

Input: Functionality, Application Footprint, Vulnerability, Attack Vector

Manual Action: None

Tool: Potential Vulnerability Identifier

Output: Potential Vulnerability, feasible Attack Vector

The identified potential vulnerabilities were sent to the development team so that the developers could consider the risks and design defense mechanisms as needed. Figure 36 shows a potential vulnerability and its attack vector against the PAL-IS "Login" functionality.

A subset of the functionalities that potentially have Stored XSS or standard SQL Injection vulnerabilities was selected for the study in this test campaign. The application footprint, the subset of functionalities, and the identified potential vulnerabilities, were documented in the test analysis document (Appendix C1).

```
Functionality: Login
Vulnerability: Standard SQL Injection
Attack Vector: Attack against password-based logon to bypass authentication
```

Figure 36 A Sample PAL-IS Functionality and Its Potential Vulnerability

5.5.3 Test Case Design

The test design followed the four major steps:

Identify Entry Points

Based on the *API for URL on PAL-IS* provided by the development team, a representative subset of the APIs that were related to the main functionalities were analyzed. The corresponding *Entry Points* together with their functionalities were enumerated and saved in the database. Figure 37 shows a sample entry point. All the entry points used in this case study are listed in Appendix C2.

```
Entry Point: Log On
ID: ep002
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Form Fields:
  username
  password
  LoginButton=Log In
Functionality: Log On
```

Figure 37 A Sample Entry Point in the PAL-IS Test Campaign

Generate Attacks

For each entry point, based on its functionality and the identified potential vulnerabilities and the feasible attack vectors, the parameter(s) in the entry point to be manipulated to create attacks were identified. The *Attacks* were then generated using a utility in the tool "Test Case Generator" which applies feasible fuzz vector payloads to the manipulated parameters of the entry points with the consideration of the application footprint:

Input: Entry Point, Potential Vulnerability, feasible Attack Vector, Fuzz Vector

Manual Action: Identify parameter(s) to be manipulate

Tool: Test Case Generator

Output: Attack

The generated attacks were saved as interim results in the database. Figure 38 shows two attacks against the entry point “Log On”.

```
Entry Point: Log On

Attack: 1
Attack Description: SQL Injection attack
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Manipulated Form Field: password=1 or 1=1--

Attack: 2
Attack Description: SQL Injection attack
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Manipulated Form Field: password=a' or 'a'='a
```

Figure 38 Sample Attacks in the PAL-IS Test Campaign

Identify Check Points

Based on the *PAL-IS System Architecture* and the discussion with the developers, two check points were identified for each attack – one check point consists of expected web page content in browser, another check point consists of an expected entry in the application logs. When we were considering a check point, we identified where a defense mechanism takes effect, where an expected result can be checked, and what was the expected result. Figure 39 shows a sample check point.

```
Entry Point: Log On

Attack: 1
Attack Description: Standard SQL Injection attack
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Manipulated Form Field: password=1 or 1=1--

Defense Mechanism: Application Specific
Check Point 1: "Login" Page
   Expected result -> the Login web page is displayed with error message: Your
login attempt was not successful. Please try again.
Check Point 2: Application log
   Expected result -> an entry in the application log with error message: A
potentially dangerous SQL injection attack was detected from the client on
login (lpeyton=1 or 1=1--)
```

Figure 39 Sample Check Points in the PAL-IS Test Campaign

The task of identifying check points was an iterative process. Initially, we relied on the information rendered in the response web page to check if an attack succeeds or not. But when we analyzed the test results of the first test round, we realized that the information in the response web page cannot provide adequate evidence to assert if an attack succeeds or not. For example, when we launched the standard SQL Injection attacks on the "Log On" page, although the logon failed (that may indicate that the attacks failed and no SQL Injection vulnerability exists), we did not have a good idea if the SQL scripts had still been executed against the database (therefore SQL Injection vulnerability might exist). This example shows that only relying on a check point in browser may lead to a false positive. On the other hand, if we treat all inappropriate error messages in response web pages as an indicator of test failures, it may lead to a false negative, as we will discuss in more detail in section 5.5.4 about the XSS false negative in the first test round. We needed to leverage the system and/or application logs as an additional check point to eliminate these false negatives/positives as they provide additional information about how the system built-in and application-specific security mechanisms trap the attacks. The design of this checkpoint in the application logs was done in collaboration between the developers and the penetration tester. The application logs were accessible as a web page returned by an HTTP request (the “application log query” showed on Figure 18). At the end, the check points in this test campaign consist of the content on response web pages at the client-side and the entries in the application logs at the server-side.

Generate Test Cases

A subset of user scenarios that includes the identified entry points were selected from *PAL-IS Use Cases (for Demo) document*, and then were saved in the database manually. For each attack, a test case was generated using the tool “Test Case Generator” (shown in Figure 18) which selects an appropriate *PAL-IS User Scenario* based on a shortest-path-scenario algorithm by which the least steps are needed to run a test:

Input: Attack, Check Point, User Scenario

Manual Action: None

Tool: Test Case Generator

Output: Test Case

Figure 40 shows a sample test case.

```
Test Case: 1
Scenario:  Login and then Logout
Step1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step2: Attack 1
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
username=lpeyton
password=1 or 1=1—
LoginButton=Log In

Step3
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Figure 40 A Sample Test Case in the PAL-IS Test Campaign

All the test design artifacts, including the *Entry Points*, *Attacks* and *Check Points*, and *Test Cases*, were documented in the test design document (Appendix C2).

5.5.4 Test Execution

In this test campaign, the tests were executed manually by entering user inputs to a web page in browser, and then observing the response web page in addition to checking the application logs on the server side. The *Test Results* were then saved in the database manually. For the test case #7 and #10, since the length of the user input (as an fuzz vector payload) exceeds the maximum limit of the field on the web page, we leveraged a HTTP proxy – WebScarab to intercept the HTTP request and then apply the fuzz vector payloads directly through the proxy to assist the test execution (shown in Figure 18).

The manual approach to test execution in this case study is due to the complexity of the PAL-IS entry points on .Net platform, which requires significantly additional effort to develop a PAL-IS specific test adaptor to load and execute the test cases automatically. Since test automation is not a major issue to be addressed in this thesis research, we decided not to pursue the PenTest workbench enhancement but instead ran the tests manually.

First Round of Testing

In the first round of test, the check point was the error message displayed on the response web page for each attack – such as “potentially dangerous form input” or “potential dangerous SQL Injection”. All the 10 test cases failed since none of the response web pages renders appropriate error message showing that the attacks had been trapped.

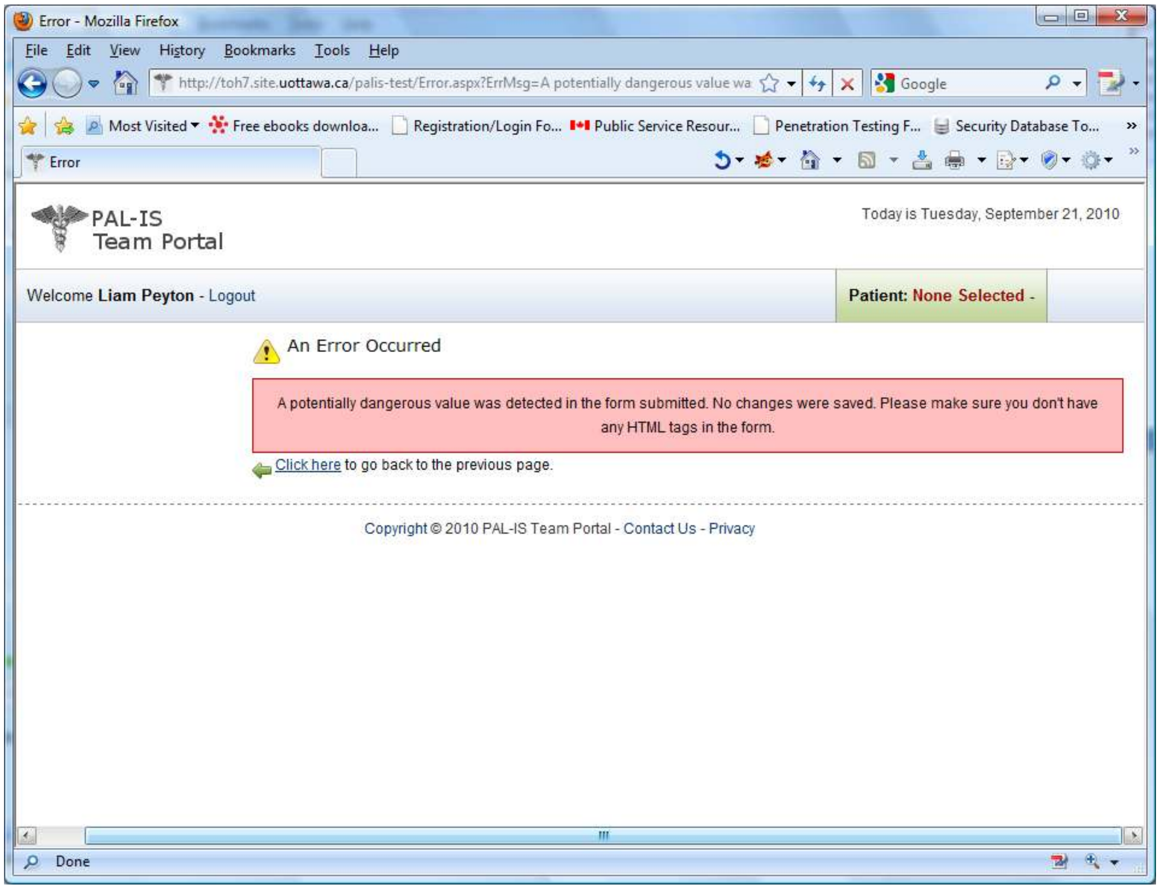
The test results were sent to the development team, and then a meeting was set between the penetration tester and the developers. The developers believed that all the attacks, including standard SQL Injection and stored XSS, should have been trapped by .Net framework (ASP.Net). The failures were false negatives and they suggested using the application logs on the server side as an additional check point. Meanwhile, the developers agreed to explicitly display appropriate error messages on the response web pages to demonstrate that the attacks have been trapped. Based on the developer's feedback, a check point in the application logs was added for each attack, and the test cases were updated accordingly.

Second Round of Testing (Regression Testing)

The same set of test cases with the updated check points were executed for the second round.

Five XSS attack test cases passed at this time – a customized error message was displayed on the response web page (Figure 41) and an error message entry was logged in the application logs (Figure 42). The failures in the first round of test were false negatives that were eliminated in this test round as a result of the more comprehensive check points.

One XSS attack was not trapped by the .Net XSS defense mechanism – no corresponding entry was found in the application logs and the user profile was updated successfully without any error message displayed (Figure 43). Similarly, all four standard SQL Injection attacks were not trapped since there were no log entries and no error messages displayed. The penetration tester was pretty sure that five true negatives were found in this testing.



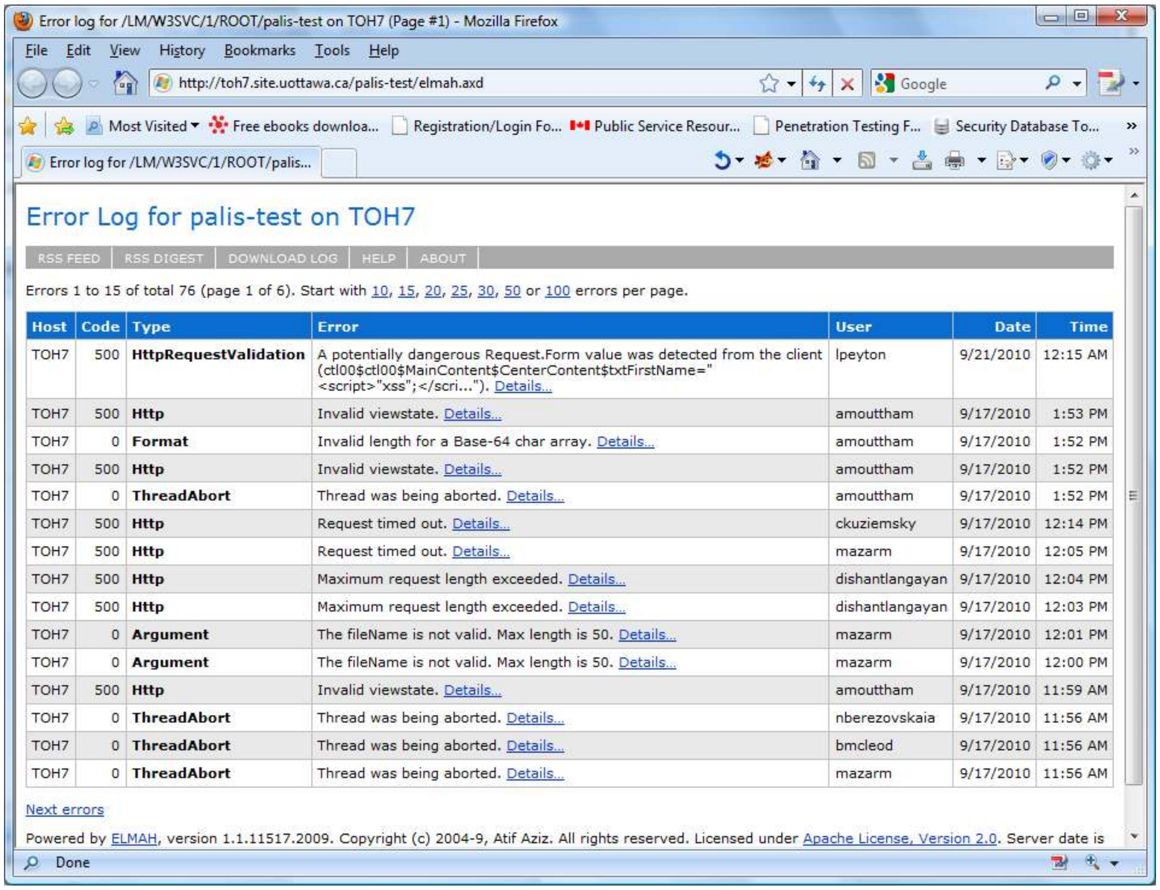
An Error Occurred

A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.

Click here to go back to the previous page.

Copyright © 2010 PAL-IS Team Portal - Contact Us - Privacy

Figure 41 Screenshot of the Error Message for an XSS Attack in the PAL-IS Test Campaign



Error log for /LM/W3SVC/1/ROOT/palis-test on TOH7 (Page #1) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://toh7.site.uottawa.ca/palis-test/elmah.axd
RSS FEED
RSS DIGEST
DOWNLOAD LOG
HELP
ABOUT
Done

Error Log for palis-test on TOH7

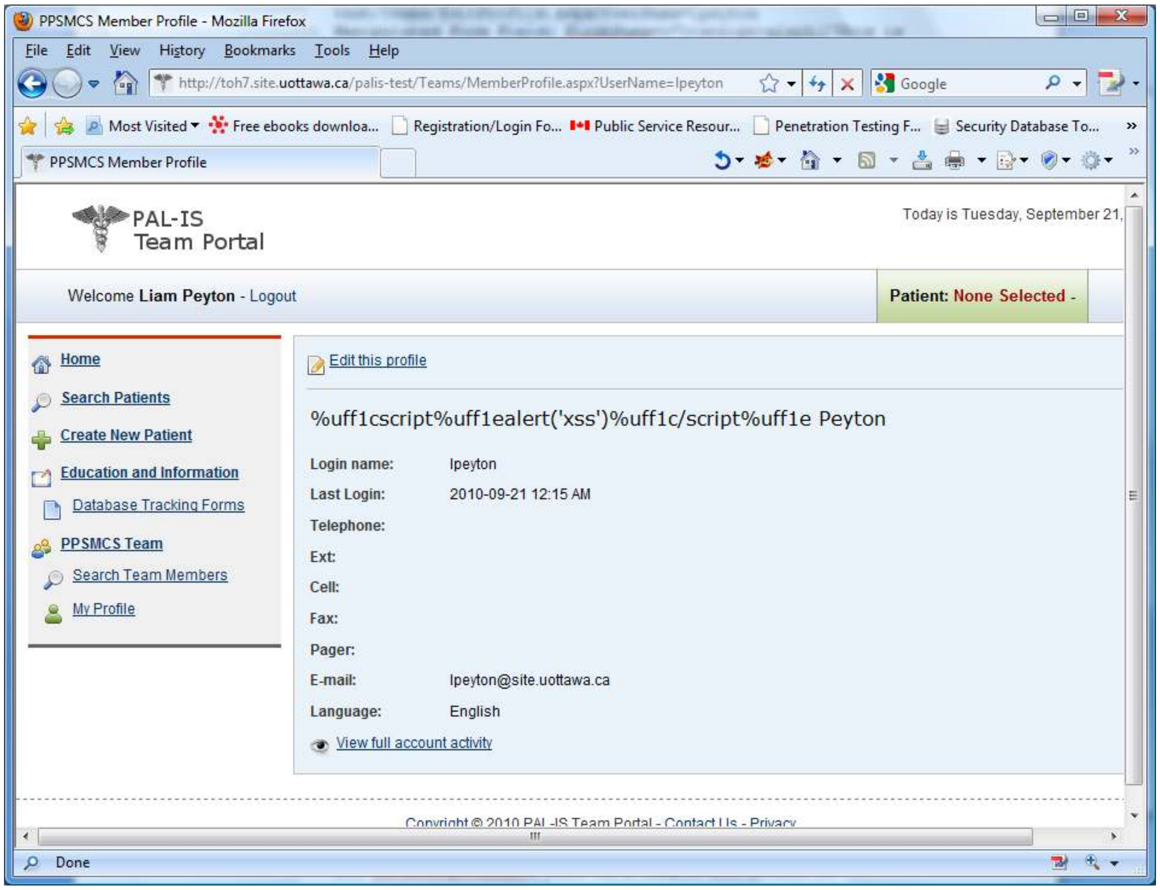
Errors 1 to 15 of total 76 (page 1 of 6). Start with 10, 15, 20, 25, 30, 50 or 100 errors per page.

Host	Code	Type	Error	User	Date	Time
TOH7	500	**HttpRequestValidation**	A potentially dangerous Request.Form value was detected from the client (ctl00$ctl00$MainContent$CenterContent$txtFirstName=" <script>"xss";</scri..."). Details...	lpeyton	9/21/2010	12:15 AM
TOH7	500	**Http**	Invalid viewstate. Details...	amouttham	9/17/2010	1:53 PM
TOH7	0	**Format**	Invalid length for a Base-64 char array. Details...	amouttham	9/17/2010	1:52 PM
TOH7	500	**Http**	Invalid viewstate. Details...	amouttham	9/17/2010	1:52 PM
TOH7	0	**ThreadAbort**	Thread was being aborted. Details...	amouttham	9/17/2010	1:52 PM
TOH7	500	**Http**	Request timed out. Details...	ckuziemsky	9/17/2010	12:14 PM
TOH7	500	**Http**	Request timed out. Details...	mazarm	9/17/2010	12:05 PM
TOH7	500	**Http**	Maximum request length exceeded. Details...	dishantlangayan	9/17/2010	12:04 PM
TOH7	500	**Http**	Maximum request length exceeded. Details...	dishantlangayan	9/17/2010	12:03 PM
TOH7	0	**Argument**	The fileName is not valid. Max length is 50. Details...	mazarm	9/17/2010	12:01 PM
TOH7	0	**Argument**	The fileName is not valid. Max length is 50. Details...	mazarm	9/17/2010	12:00 PM
TOH7	500	**Http**	Invalid viewstate. Details...	amouttham	9/17/2010	11:59 AM
TOH7	0	**ThreadAbort**	Thread was being aborted. Details...	nberezovskaia	9/17/2010	11:56 AM
TOH7	0	**ThreadAbort**	Thread was being aborted. Details...	bmcleod	9/17/2010	11:56 AM
TOH7	0	**ThreadAbort**	Thread was being aborted. Details...	mazarm	9/17/2010	11:56 AM

Next errors

Powered by ELMAH, version 1.1.11517.2009. Copyright (c) 2004-9, Atif Aziz. All rights reserved. Licensed under Apache License, Version 2.0. Server date is

Figure 42 Screenshot of the Application Logs in the PAL-IS Test Campaign



PPSMCS Member Profile - Mozilla Firefox

http://toh7.site.uottawa.ca/palis-test/Teams/MemberProfile.aspx?UserName=lpeyton

PAL-IS Team Portal

Today is Tuesday, September 21,

Welcome **Liam Peyton** - Logout

Patient: None Selected -

Edit this profile

%uff1cscript%uff1ealert('xss')%uff1c/script%uff1e Peyton

Login name:	lpeyton
Last Login:	2010-09-21 12:15 AM
Telephone:	
Ext:	
Cell:	
Fax:	
Pager:	
E-mail:	lpeyton@site.uottawa.ca
Language:	English

View full account activity

Copyright © 2010 PAL-IS Team Portal - Contact Us - Privacy

Done

Figure 43 Screenshot of a Successful XSS Attack in the PAL-IS Test Campaign

Another meeting was then set up. The developers and the penetration tester analyzed the failed test cases together:

For the failed test case #7 (a Stored XSS attack), while in general .Net 3.5 SP1 provides a built-in defense mechanism against stored XSS attacks, it fails to trap a specific fuzz vector (XSS vulnerabilty in ASP.Net, 2005) that uses an obscure character encoding. Microsoft did not consider it a serious security issue, so they listed it as a known issue but did not plan to fix it (Widescale Unicode Encoding Implementation Flaw Discovered, 2007). Therefore, the development team decided it was not a priority to fix too.

For the failed SQL Injection test cases, it was found out that .Net 3.5 SP1 itself does not provide any defense mechanism against standard SQL Injection attacks. ibatis, a third-party component that was used in AEMS, provides such protection to an application. However, ibatis was not used in the current PAL-IS design. The development team decided to develop an application specific defense mechanism against the standard SQL Injection attacks. They introduced a hand-crafted validator that carefully validates user input, e.g. password and patient searches, and rejected input that was not of the correct format and length, in particular rejecting user input that contained spaces. This follows the Microsoft Guidelines about SQL Injection (Microsoft Security and Protection: SQL Injection, 2011) to protect PAL-IS from the standard SQL Injection attacks used in this test campaign.

Third Round of Testing (Regression Testing)

When the implementation of the defense mechanism was completed, the same set of test cases was executed to validate the fix and exclude any regression caused by the fix. At this time, the four SQL Injection test cases passed.

Based on the security analysis during the test executions, we realized that it is important to document the third-party components that play a security role in a web application. The test campaign model was then refined by adding a new attribute in *Check Point* to specify the defense component that traps attacks, and a new attribute in *Application Footprint* to explicitly represent third-party security components used in a web application. This security knowledge can help developers to understand which security vulnerabilities are not protected in the web application so that they need to consider application specific security measures. It can also help penetration testers to identify additional check points to determine if attacks are trapped or not. In addition, the information of third-party components can facilitate test maintenance and regression testing. For example, in a later release if ibatis is integrated into PAL-IS, then it is straightforward to identify an additional check point for the SQL Injection test cases. Or when the XSS issue in ASP.Net is fixed in the future .Net version and PAL-IS is upgraded to this new version, the relevant test cases can be executed to verify if the attacks are trapped this time.

When the test execution was completed, the tool “Test Report” was run to generate and print out the *Test Report* based on the test results. The test report was documented in the test execution document (Appendix C3). Figure 44 shows a sample test report.

```
------------- Test Campaign Summary --------------
# of test cases executed: 10
# of failed tests: 5
# of identified vulnerabilities: 2
# of entry points tested: 8
# of vulnerable entry points: 3
--------------------------------------------------

------- Web Security Repository Statistics -------
# of vulnerabilities: 5
# of attack vectors: 9
# of fuzz vectors: 9
--------------------------------------------------

----------------Test Result Details---------------
Test Case 1: launch an SQL Injection attack against <Log On> page
Result: Failed

Test Case 2: launch an SQL Injection attack against <Log On> page
Result: Failed

Test Case 3: launch an SQL Injection attack against <Search Patient> page
Result: Failed

Test Case 4: launch an SQL Injection attack against <Search Patient> page
Result: Failed

Test Case 5: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 6: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 7: launch an Stored XSS attack against <Update Profile> page
Result: Failed

Test Case 8: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 9: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 10: launch an Stored XSS attack against <Update Profile> page
Result: Pass

--------------------------------------------------

------- Identified Vulnerability Details ---------

Vulnerability: SQL Injection
probably exists on:
  Entry Point: ep002(Log On)
  Entry Point: ep005(Search Patient)

Vulnerability: Stored XSS
probably exists on:
  Entry Point: ep008(Update Profile)
--------------------------------------------------
```
Figure 44 A Sample PAL-IS Test Report

5.5.5 Test Result Summary

There were three iterations of text execution using the same set of test cases in this test campaign. Table 6 shows the PAL-IS test campaign summary.

Table 6 The PAL-IS Test Campaign Summary

Web Security Knowledgebase	
Vulnerability	5
Attack Vector	9
Fuzz Vector	26
Test Effort	
Entry Points	8
Potential Vulnerable Entry Points	3
Test Cases	10
Test Result	
First round	10 test cases failed, 2 types of vulnerabilities were identified on 3 entry points
Second round	5 test cases failed, 2 types of vulnerabilities were identified on 3 entry points
Third round	1 test cases failed, 1 vulnerability (known MS XSS) was identified on 1 entry point

Chapter 6. Evaluation

The evaluation of our framework is largely based on the three case studies presented in Chapter 5. In section 6.1, we discuss the intended use of the proposed penetration test framework as well as the categorization of the case studies in terms of how representative they are.

In section 6.2, we evaluate the framework by comparing it with the Microsoft Security Development Lifecycle (SDL) as followed by typical small .NET development teams. The criteria identified in section 4.2, are the focus of that evaluation.

In section 6.3, we compare the framework with the other security assurance approaches mentioned in section 3.4. We use the major high level criteria from section 3.2 in the comparison (integrated penetration test methodology, feasible for general testers, and systematic test campaign management), but also consider the more basic common criteria that are usually used to classify these other approaches.

In section 6.4, we discuss this complementary nature of our framework, to indicate that the greatest value of our framework is not as an alternative to other approaches but as a complement to them.

In section 6.5, the WenbGoat case study is further analyzed with respect to the representative and the effectiveness of the test campaign.

In section 0, we analyze the limitations in this thesis research, including the thesis research methodology, the generic software development model and the simplified web application architecture based on which the test framework is proposed, the simple web security knowledgebase used in the case studies, and the case studies themselves.

6.1. Intended Use of Framework and Case Studies

The proposed test framework is not intended to be used as a "one solution fits all". Neither is it intended to be used to replace the expert security assessment. Actually, security experts should be leveraged to build security knowledgebase for penetration test campaigns, and they may be invited to help develop adequate and appropriate security remediation measures for security defects identified in penetration test campaigns. It is intended to assist general testers, who are responsible for penetration testing in the context of a security-oriented software development life cycle, by providing them with a well-defined test methodology supported by a model-driven penetration test approach in which the security knowledgebase, the utilized development artifacts, and the produced test artifacts are represented declaratively so that they can be processed by computer programs to automate some aspects of the testing.

Considering the assumptions based on which the proposed test framework was developed (section 4.3), the framework is intended to apply to penetration test campaigns, which are conducted in a classic iterative software development life cycle where development artifacts and developers' assistance are available to penetration testers, for web applications that are developed based on a classic AJAX architecture that do not process multimedia content and do not have interacts with plug-ins/add-ons in browser that have security implications.

Two of the three case studies (AEMS and PALIS) were developed in a classic iterative software development life cycle. They are significant but relatively small web applications developed by small .NET teams. However, in the three case studies, the test campaigns are quite different with respect to the development phase when the test campaign got started and the application footprint.

WebGoat is an existing reference web application that was useful for testing our PenTest workbench and validating our web security knowledgebase and penetration test campaign model. However, our test campaign for WebGoat was not integrated into its software development life cycle and there was no interaction between testers and developers. AEMS is a real web application that was still in development, approaching its beta release, when our penetration test campaign started. As a result, we did not integrate over the entire software development lifecycle and some development artifacts were not well documented. PAL-IS is a real web application that was just beginning development when our penetration test campaign s started. As a result, our complete test methodology was applied across an entire software development lifecycle and the development effort benefited from fruitful collaboration between the development team and the test campaign.

The framework includes a generic, knowledge-based prototype PenTest workbench to support the test campaigns in the three case studies in which the three applications have different application footprints. WebGoat is a J2EE application that uses HTTP base authentication and can be accessed via HTTP, while both AEMS and PAL-IS are .NET applications, use form-based authentication, and AEMS was configured to be accessed over HTTPS. The same PenTest workbench was used to automate the test tasks including the generation of potential vulnerabilities, attacks, and test cases in all the three test campaigns. It was also used to load and run test cases automatically in the WebGoat and AEMS test campaigns. The test cases for PAL-IS test campaigns were executed manually, as justified in section 5.5.3. We believe, though, with extra effort to develop a PAL-IS specific test adaptor, the PenTest workbench can be enhanced to load and run the PAL-IS test cases automatically.

6.2. Compare with the MS SDL

As shown in Table 7, based on our case studies, we evaluate the proposed test framework using the evaluation criteria defined in section 4.2, by comparing the results obtained in our case studies using the Microsoft Security Development Lifecycle (MS SDL) on its own, versus results obtained using the MS SDL integrated with support from our test framework in a systematic manner. We want to use the evaluation to demonstrate our framework, as a complement to the MS SDL, how it enhances penetration testing practice in such a security-oriented software development life cycle. We do not compare our framework with the OWASP testing guide since the OWASP testing guide is primarily to define what kinds of attacks a penetration tester needs to do, rather than what tasks a tester needs to perform in each phase of software development life cycle and how they should collaborate with developers and security experts.

The evaluation is based on what we observed in the case studies, because in this thesis research we are simply evaluating whether the framework is feasible for general testers, rather than to the level of what is the best design of such a framework.

Each criteria is evaluated on a scale of *Yes*, *Ad Hoc*, and *No*, where *Yes* means the criterion is addressed in a systematic fashion, *Ad Hoc* means it is possible to address the criterion but how to do so is not well-defined and is very much dependent on the style and expertise of a tester or security expert, and *No* means the criterion is not addressed at all. Abbreviations are used in Table 7 for better readability: MDPT stands for the proposed *Model-Driven Penetration Test framework*, and MS SDL stands for the *Microsoft Security Development Lifecycle*.

Table 7 Comparison with the MS SDL

	MDPT-integrated MS SDL	Standalone MS SDL
Integrated Test Methodology		
Integrated Penetration Test Process	Yes, Well Defined	No
Collaboration with Developers	Yes, Well Defined	Ad Hoc
Development Artifacts Utilization	Yes, Well Defined	Ad Hoc
Grey-Box Test Architecture	Yes (Simple one)	Ad Hoc
Use with Other Security Test Methods	Ad Hoc	Ad Hoc
Feasible for General Testers		
Web Security Knowledgebase	Yes (Prototype, Limited)	Ad Hoc
Test Campaign Modeling	Yes, Well Defined	No
Workbench Tool Support	Yes (Prototype)	Ad Hoc
Consistent Test Coverage	Yes, Well Defined	Ad Hoc
Systematic Test Campaign Management		
Reports and Documentation	Yes (Simple Templates)	Ad Hoc
Test Maintenance	Yes, Well Defined	Ad Hoc
Regression Testing	Yes, Well Defined	Ad Hoc

Integrated Penetration Test Process

- MDPT-integrated MS SDL: It is a penetration test process completely integrated into a security-oriented software development life cycle. Vulnerabilities identified in a test campaign can be addressed by developers before an application is released.

In the PAL-IS test campaign, the tester got involved in the very beginning of the development of PAL-IS. The test campaign was fully integrated with the application development. The test campaign was developed starting from the initial system analysis to prototype design to alpha testing and then beta testing of the application. In the AEMS test campaign, the tester got involved in the middle of the development of AEMS (a beta version) and went through test development and beta testing. The vulnerabilities identified in the two test campaigns got addressed by the developers before the web application is released to production. The fixes were validated promptly and the regression testing was also performed.

- Standalone MS SDL: In both the PAL-IS and the AEMS development, the developers, who more or less followed the MS SDL in their development process, left the penetration testing as an afterthought and were to rely on external security experts to come in and test the web application after development was complete before they were going into full production.

Collaboration with Developers

- MDPT-integrated MS SDL: It facilitates the collaboration between developers and penetration testers by explicitly specifying the roles of developers and penetration testers, the touch points, and the development and test artifacts involved. Both developers and penetration testers can benefit from the collaboration.

  In the PAL-IS test campaign, at the end of the analysis phase, the penetration tester sent a list of potential vulnerabilities to the developers who were then aware of the potential vulnerabilities (Cross-Site Scripting and Standard SQL Injection). They thought that the .Net platform provides build-in security mechanisms to protect the application from the vulnerabilities.

  Before the second round of test execution in the PAL-IS test campaign, the developers assisted the penetration tester to enhance the check points by adding a check point in the application logs for each test case. As the result, five false-positives in the first round of test execution were eliminated in the second round of test execution.

Furthermore, the process of adding the additional check points helped the developers get a better understanding how the defense mechanisms work – with the four failed Standard SQL Injection test cases in the second round of test execution, the developers realized that the .Net platform itself does not provide a built-in defense mechanism against Standard SQL Injection attacks. As a result, the developers developed an application-specific defense mechanism to remedy the Standard SQL Injection vulnerability.

- Standalone MS SDL: It is possible, but not required for security experts to work with developers to facilitate their penetration test effort, but there is no well-defined process to follow.

Development Artifacts Utilization

- MDPT-integrated MS SDL: It specifies the development artifacts that can be utilized at each test phase. The corresponding test artifacts can be derived from the development artifacts directly.

  In the PAL-IS test campaign, at the test analysis phase, the tester derived the test artifacts *Application Footprint* and *Functionality* from *PAL-IS Project Specification (Section 4.2 – Functionality)* and *PAL-IS Use Cases (for Demo)*. At the test design phase, the tester enumerated *Entry Point* from *API for URL on PAL-IS*; identified *Check Point* based on the *PAL-IS Project Specification Document (Section 5 – Architecture)*; and retrieved *User Scenario* from *PAL-IS Use Cases (for Demo)* to generate *Test Case*.

- Standalone MS SDL: Security experts may leverage development artifacts in penetration testing, but there are no development artifacts that have been explicitly specified to be used in a test campaign.

Grey-Box Test Architecture

- MDPT-integrated MS SDL: It defines a grey-box test architecture that explicitly specifies possible entry points and check points that can be used in a test campaign. The grey-box test architecture discussed in the case studies was developed based on simple AJAX web applications that only have HTML pages and limited client-side scripts.

  In the PAL-IS test campaign, the check points in the application logs were created and used in the second round of test execution so that the five false-positives in the first round of test execution were eliminated effectively (see the test result analysis in section 5.4.5 for more details).

- Standalone MS SDL: Security experts may use the knowledge of internal structure of web applications in penetration testing, but there is no grey-box test architecture defined that explicitly specifies possible entry points and check points that can be used in a test campaign.

Use with Other Security Test Methods

- MDPT-integrated MS SDL: It is not a major issue we address in the proposed test framework. We think with the well-defined collaboration between developers and testers, other security test methods can be used to enhance the penetration test effort, e.g. confirm the vulnerability findings in a test campaign.

  For example, in the AEMS test campaign, the developers did code reviews with the tester for the false negative of Standard SQL Injection vulnerability (see section 5.4.4 for more details).

- Standalone MS SDL: Security experts may use other security test methods with their penetration test effort to improve test quality or test efficiency, but there is no well-defined process to follow.

Web Security Knowledgebase

- MDPT-integrated MS SDL: It defines a simple, relational database based web security knowledgebase that was populated with limited but common types of vulnerabilities and related attack vectors and fuzz vectors. The web security knowledgebase was used directly in the test campaigns to support test development.

  The same web security knowledgebase was used in all the three case studies. The penetration tester generated the test artifacts *Potential Vulnerability*, *Attack*, and *Test Case* by using the PenTest work bench that interacts with the web security knowledgebase to retrieve the required security knowledge. The penetration tester did not need to worry about the updates and maintenance of the security knowledgebase.

- Standalone MS SDL: Security experts may have their own web security knowledgebase, or use tools that come with knowledgebase but none are provided or defined by MS SDL. Typically their format is not standardized nor are they typically made available to development teams to use as a resource for learning and reuse.

Test Campaign Modeling

- MDPT-integrated MS SDL: It defines a test campaign model that captures and retains the test artifacts and the development artifacts involved in penetration test campaigns.

  The same test campaign model was used in all the three case studies. During the test development, the penetration tester focused on populating the model by using the development artifact e.g. *User Scenario* and the test artifacts e.g. *Functionality*, *Entry Point*, *Check Point* etc., and then used the utilities in the PenTest workbench to generate the test artifacts e.g. *Potential Vulnerability*, *Attack*, *Test Case* etc. During the test maintenance, the modeled artifacts were reused to update the impacted test artifacts. The test campaign model is also important for enabling the tool support that was provided in the prototype workbench.

- Standalone MS SDL: No test or development artifacts are modeled for penetration testing.

Workbench Tool Support

- MDPT-integrated MS SDL: A prototype of knowledge-based PenTest workbench, which consists of a small set of model-driven utilities, was developed to support the proposed penetration test tasks at each test phase. The workbench provides an integrated test platform that streamlines the test effort in a way that an output of a utility can be used as an input of another utility at the next step.

  The PenTest workbench was used in all the three test campaigns to generate the potential vulnerabilities at the test analysis phase, the attacks and the test cases at the test design phase, and the test execution reports at the test execution phase. In the WebGoat test campaign and the AEMS test campaign, the test cases were loaded and executed automatically using the PenTest workbench.

- Standalone MS SDL: There are some tools available for web application penetration testing that security experts can leverage based on their own choice. But these tools are not built on an integrated test platform so that the security experts have to interpret and transform the output of one tool to input of another tool which can be time-consuming and error-prone.

Consistent Test Coverage

- MDPT-integrated MS SDL: It defines a systematic process, with tool support, to achieve consistent test coverage across test campaigns, in terms of the coverage of entry points and the coverage of potential vulnerabilities.

In the PAL-IS test campaign, the web security artifact *Functionality* that was used to identify the *Potential Vulnerability* was derived from the development artifact *PAL-IS Project Specification Document (Section 4.2 – Functionality)*. The test artifact *Entry Point* was derived from the development artifact *API for URL on PAL-IS*. In the AEMS test campaign, the similar development artifacts were acquired from the developers and the relevant test artifacts were derived from the development artifacts in the same systematic way.

In all the three test campaigns, the same web security knowledgebase, which determines the types of vulnerabilities that can be uncovered, were used to generate the test artifacts *Potential Vulnerability* and *Attack* by using the PenTest workbench.

- Standalone MS SDL: Test coverage is determined by the expertise and diligence of security experts. The test quality may vary from one security expert to another.

Reports and Documentation

- MDPT-integrated MS SDL: It defines a set of simple templates that can be used for test documentation at the test analysis, test design, and test execution phase. The standard test documents can facilitate the communication between developers and testers at the various phases.

  The same test templates were used in all three case studies to record test artifacts and test results. The test execution reports were generated automatically based on the test results by using the PenTest workbench.

- Standalone MS SDL: Security experts may have their own test document templates. The test templates may vary from one expert to another.

Test Maintenance

- MDPT-integrated MS SDL: It defines when and how to perform test maintenance in test campaigns. As the test artifacts and development artifacts are modeled and processed by computer programs, they can be reused across multiple test campaigns on a continuous base.

  In both the AEMS and the PAL-IS test campaigns, after the first round of test execution, only the test artifact *Check Point* was updated due to the fixes in the implementation that addressed the failures in the first round of test execution and the enhancement in the check points (additional check points were added for each attacks). Other test artifacts, such as *Entry Point* and *Attack*, were reused to re-generate test cases. And then the updated test cases were executed in the second round of test execution, and the third round of test execution in the PAL-IS test campaign.

- Standalone MS SDL: Penetration testing is an optional, one-time task so that there is no consideration for test maintenance. Security experts may have their own way to manage test maintenance that may vary from one expert to another, but there is no well-defined methodology that can be followed.

Regression Testing

- MDPT-integrated MS SDL: It defines when regression testing is required. Since test artifacts (such as test cases) are modeled and can be processed by computer programs, regression testing can be automated.

  In the AEMS test campaign, in the second round of test execution the updated test cases were loaded and executed automatically using the PenTest workbench to ensure the failure (Standard SQL Injection attack against the entry point ep006) in the first round of test execution was fixed and there was no regression.

In the PAL-IS test campaign, the updated test cases were manually executed in the second round of test execution to ensure the failures (Stored XSS attack against the entry point ep008) was fixed and there was no regression. The same test cases were manually executed in the third round of test execution to ensure the failures (Standard SQL Injection against the entry points ep002 and ep005) were fixed and there was no regression.

- Standalone MS SDL: Penetration testing is an optional, one-time task so that there is no consideration for regression testing. Security experts may have their own way to manage regression testing that may vary from one expert to another, but there is no well-defined methodology that can be followed.

6.3. Comparison of Security Assurance Approaches

As shown in Table 8, we compare the framework with the other security assurance approaches mentioned in section 3.4, including Code Reviews that systematically examines source code of a web application to improve the overall quality of the application, Code Analyzer that scans source code of a web application to discover weaknesses in the code, Vulnerability Scanner that scans a running web application for potential vulnerabilities; and Expert Security Assessment as described in section 3.3.2. We use the major high level criteria from section 4.2 in the comparison (integrated penetration test methodology, feasible for general testers, and systematic test campaign management), but also consider the more basic common criteria that are usually used to classify these other approaches, including:

- <u>Coverage of Entry Points/Code Base</u>: is it feasible to systematically achieve coverage of all entry points or code base
- <u>Coverage of Vulnerability Types:</u> is it feasible to consistently identify various types of vulnerabilities

- Test Automation: is test automation supported

Table 8 Comparison of Security Assurance Approaches

	MDPT	VS	CR	CA	ESA
Common Criteria					
Coverage of Entry Points/Code Base	Yes	Yes	To Some Degree	Yes	Yes
Coverage of Vulnerability Types	Yes	To Some Degree	Yes	To Some Degree	Yes
Test Automation	To Some Degree	Yes	No	Yes	To Some Degree
Categories of the Evaluation Criteria					
Integrated Test Methodology	Yes	No	Yes	No	No
Feasible for General Testers	Yes	Yes	No	No	No
Systematic Test Campaign Management	Yes	To Some Degree	No	To Some Degree	No

The security assurance approaches are compared on a scale of *Yes*, *To Some Degree*, and *No*, where *Yes* means an approach completely meets a criterion, *To Some Degree* means it meets a criterion partially, and *No* means it does not meet a criteria at all. Abbreviations are used in Table 8 for better readability: MDPT stands for the proposed *Model-Driven Penetration Test framework*, VS stands for *Vulnerability Scanner*, CR stands for *Code Reviews*, CA stands for *Code Analyzer*, and ESA stands for *Expert Security Assessment*.

Coverage of Entry Points/Code Base

- MDPT: Entry points are enumerated based on design documents such as Web API in a systematic way, as what was done in the PAL-IS test campaign.

  An implementation of a web application may contain entry points that are not specified in the design document, and vulnerabilities may exist in these entry points. We did not address this issue in the proposed test framework. We think tools such as a web spider (crawler) can be leveraged at the implementation phase to gather all existing entry points and identify any ones that are not specified in the design document.

- VS: Vulnerability scanner can browse a web application and identify all existing entry points. It may fail, though, to identify any entry points that are specified in the design document but missed in the implementation of the web application.
- CR: Code reviews is a time consuming task that is usually not applied to the entire code base of a web application with a typical scale.
- CA: Code analyzer can be used to scan the entire code base of a web application to identify insecure coding practice.
- ESA: It is assumed that security experts can examine all possible entry points in a web application within reasonable time frame by leveraging their expertise and with tool support.

Coverage of Vulnerability Types

- MDPT: The test campaigns are developed and managed in a systematic, model-driven approach with tool support. Security experts are expected to create and maintain a web security knowledgebase that retains their security expertise and determines the types of vulnerabilities that can be identified in a test campaign. In the three case studies, the key test artifacts, specifically Test Case and Test Result (identified vulnerabilities), can be consistently reproduced by applying the test framework.

  A simple web security knowledgebase, which includes limited types of vulnerabilities, was defined and used in the case studies. The effort required and difficulty of creating a large complete security knowledgebase is a major issue for our test framework and is a significant issue that needs to be researched in future work.

- VS: Vulnerability scanner is efficient in identifying some specific types of vulnerabilities in a web application but it is not good at finding other types that are more complex or application specific.
- CR: Code reviews can find all types of insecure design and coding practice when it is performed by people with adequate security expertise, e.g. security experts.
- CA: Code analyzer is efficient in finding some specific types of insecure coding practice but is not good at finding all types of insecure coding.
- ESA: It is assumed that security experts can identify all types of vulnerabilities in a web application with their expertise and with tool support.

Test Automation

- MDPT: Some test artifacts, including Potential Vulnerability, Attack, and Test Case, can be generated automatically using the PenTest workbench based on other development artifacts, test artifacts and the web security knowledgebase, as demonstrated in the three case studies. In addition, the test artifacts can be reused across multiple test campaigns.
- VS: In general, it is a test automation tool. It may require significant effort to set it up before running.
- CR: In general, it is a manual test process.
- CA: In general, it is a test automation tool.
- ESA: Security experts utilize a variety of tools to assist in the security assessment. However, the tools are not built into an integrated test environment so that the output of one tool has to be manually interpreted and transformed to the input of another tool.

Integrated Test Methodology

- MDPT: It provides a test methodology that is completely integrated into a security-oriented software development life cycle.

- VS: It is run against a deployment of web application to identify any vulnerability. It can be integrated into a security-oriented development life cycle at a specific phase, e.g. verification phase. But this is not a mandatory usage of vulnerability scanners.
- CR: It can be performed as an integrated process in a security-oriented software development life cycle.
- CA: It is run against source code of a web application to identify any insecure coding practice. It can be integrated into a security-oriented development life cycle at a specific phase, e.g. application implementation phase. But this is not a mandatory usage of code analyzers.
- ESA: It is usually performed by security experts as an isolated, one-time test process that is not considered in the context of a security-oriented software development life cycle.

Feasible for General Testers

- MDPT: General testers are capable of performing penetration test campaigns on a continuous base by taking a model-driven test approach that is supported by the PenTest workbench and the web security knowledgebase.
- VS: General testers can utilize it for security assurance, although special training may be required.
- CR: Usually, it is performed by developers with adequate security expertise or by security experts themselves.
- CA: Developers utilize it for ensuring secure coding practice.
- ESA: It is performed by security experts, external or internal.

Systematic Test Campaign Management

- MDPT: The integrated test methodology provides guidance on when and how test maintenance and regression testing are required. The model-driven test approach enables the created test artifacts can be reused across multiple test campaigns.

- VS: It provides the feature of test artifact management and supports regression testing. It does not, though, provide a systematic guidance for test artifact maintenance and regression testing.
- CR: In general, it does not provide systematic support for regression testing.
- CA: Usually, a code analyzer is used to scan source code that does not need test maintenance.
- ESA: In general, it is a one-time task without consideration for test maintenance and regression testing.

6.4. Complementary Nature of the Proposed Test Framework

We have been comparing our framework to other approaches. However, the most significant potential utility of our framework is not as an alternative to other approaches but rather as a complement that supports them effectively. In the sub-sections below, we illustrate the complementary nature of the test framework by discussing how the test framework can be used with the MS SDL and with vulnerability scanners.

6.4.1 Supplement to the Microsoft Secure Development Lifecycle

The Microsoft Security Development Lifecycle (SDL) defines a security-oriented software development life cycle with tool support for each phase. In the MS SDL, penetration testing is an optional task that is performed by external security experts at the verification phase.

Both the proposed test framework and the MS SDL propose an integrated security assurance process. The MS SDL was developed from the development perspective and leverages a range of security assurance methods such as threat modeling, static analysis and dynamic fuzz testing, but it does not define "how" the penetration testing should be performed. The proposed test framework only focuses on penetration testing and specifies a detailed test methodology with tool support. The framework can be used as a supplement to the MS SDL in that it provides detailed guidance to general testers to perform systematic penetration test campaigns, including:

- Defined a test process that is fully integrated into a security-oriented software development life cycle, such as the MS SDL, with explicitly specified touch points between developers and testers, and specified the development artifacts that can be used for test development.
- Defined three basic roles in a penetration test campaign, namely security expert, developer, tester, and their responsibilities.
- Defines a model-driven penetration test approach based on the web security knowledgebase and test campaign model.

6.4.2 Use Vulnerability Scanner in the Test Framework

Web application vulnerability scanners provide an automated approach to security assurance. Vulnerability scanners alleviate the requirements of security expertise to testers. General testers can run a scanner to uncover potential vulnerabilities existing in a web application, and the scanning can be repeated easily. Scanners are very efficient at finding some specific types of vulnerabilities, e.g. denial of service; redundant backup files; or path traversal vulnerability. However, scanners can’t find all types of vulnerabilities, especially those deeply intertwined in business logic and custom application design. Therefore, testers can leverage a scanner as a supplementary tool in the proposed test framework in that they can use it to scan for the specific types of vulnerabilities that it is really good at, and get themselves focus on the types of vulnerabilities that are more relevant to application specific flaws.

6.5. WebGoat Reference Analysis and Performance

The WebGoat application used in the case study is version 5.2. It includes 55 "attack lessons"[3] – each lesson is to demonstrate an attack scenario (against a deliberately planted security defect in WebGoat), or just explain an attack method (e.g. how to browse a WSDL file for a published web service to find all operations provided in the web service), or simply show possible weakness in a system (e.g. the strength of various password scheme). Some attacks are pretty straightforward, e.g. using WebScarab to intercept and alter an HTTP request or even simply adding "&admin=true" to the URL in browser to bypass access control scheme, or simply review HTML source code to look for any sensitive information. Some attacks are trickier, e.g. SQL Injection attacks and XSS attacks that require dedicatedly manipulating specific parameters. The 55 lessons cover various types of web application vulnerabilities, such as Cross-Site Scripting, Injection flaws (e.g. standard SQL Injection), session management flaws, access control flaws, authentication flaws, AJAX specific vulnerabilities, or web service specific vulnerabilities.

I went through and exercised all the lessons during the WebGoat case study. Then I picked up the defects in four lessons: a reflected XSS, a stored XSS, a numeric SQL Injection, and a string SQL Injection, based on the considerations that the four attack vectors are "complicated" ones and they are the types of vulnerabilities in the sample web security knowledgebase.

[3] Based on (OWASP WebGoat User and Install Guide, 2011), WebGoat includes around 30 "lessons". But we found 55 lessons in the version we installed and used.

The WebGoat test includes 13 entry points, 4 user scenarios, and 13 generated test cases. All four defects were uncovered by the tests successfully (true negative), and the test did not produce any false negative and false positive. See section 5.3 and Appendix A1 – A3 for details. Below are the major steps and related effort that I took to run the test campaign by applying the test framework:

- Identified the footprint of WebGoat and populated it into the test campaign database manually: 2 hours
- Identified 13 entry points (and its functionality) using WebScarab and populated it into the test campaign database manually: 4 hours
- Identified the parameter(s) to be manipulated for each entry point that has the security defect, and populated it into the test campaign database manually: 2 hour
- Generated 13 attacks using the PenTest workbench: mere seconds for each attack, plus a few minutes to double check the result, once we had a bug-free implementation of our PenTest workbench. (But it did take several weeks of coding effort before the PenTest workbench was finalized).
- Identified check points for each attack by referring to the solution notes in WebGoat, and populated it into the test campaign database manually: 1 hour
- Identified 4 user scenarios for the attacks by walking through WebGoat and capturing relevant HTTP requests using WebScarab, and populated it into the test campaign database manually: 2 hours
- Generated 13 test cases using the PenTest workbench: seconds for each test case, plus a few minutes to double check the result, once we had a bug-free implementation of our PenTest workbench. (But it did take several weeks of coding effort before the PenTest workbench was finalized).
- Ran the test cases using the PenTest workbench and analyzed the test results: seconds to run the test cases, but several hours (~ 3 hours) were spent verifying and validating that the results were correct.

Table 9 shows the summary of the discussions above.

Table 9 WebGoat Reference Analysis and Performance

# lessons	# defects	# tested defects	# test cases	false negative/positive	test effort
55	40	4	13	none	14 hours

6.6. Limitations

There are some limitations to the results obtained in our thesis that should be acknowledged in terms of the research methodology applied in this thesis, the proposed test framework, and the case studies used for the evaluation of the framework.

6.6.1 Design-Oriented Research Methodology

A design-oriented research methodology was followed in this thesis research (as was described in section 1.4). By conducting a literature research and gap analysis, we identified a set of outstanding issues existing in current penetration test practice. Then we designed and proposed a test framework to address these specific issues (namely the criteria identified in section 4.2). However, there may be other issues and criteria that are relevant (for example the ones listed in section 6.3), which may not be present in our framework. We have not attempted to validate in a comprehensive fashion that our framework is better in all ways for testing of all web applications. Instead, we focused on the issues that we identified, and evaluated and demonstrated that the proposed test framework can be better than other approaches with respect to the identified issues for certain types of web applications by conducting case studies and problem analysis. A more comprehensive evaluation of our framework across a much wider spectrum of web applications and development teams is needed, but that is beyond the scope of this thesis.

6.6.2 Software Development Life Cycle Model

The test methodology in the proposed test framework was developed based on the assumption (see the assumption 1 in section 4.3) that the development of web applications under test follows a classic iterative software development process. However, there are a variety of software development life cycle models available that web application development may follow, e.g. less formal agile methodology or more complex and structured RUP approach. In these different development processes, the available development artifacts that are relevant to penetration testing and the possible interactions between developers and penetration testers may be different. The test methodology in the proposed test framework would need to be amended to adapt to such situations.

6.6.3 Web Application Architecture

In the proposed test framework, the grey-box test architecture was developed based on classic AJAX web applications that do not process multimedia content and do not interact with any plug-ins/add-ons in browser (see the assumption 2 in section 4.3). The grey-box test architecture, including the possible entry points and check points, and the consequently developed test campaign model, are discussed based on this simplified architecture. However, many web applications have a more sophisticated architecture that is capable of processing multimedia content and can interact with various plug-ins/add-ons components in browser. These web applications may have different types of vulnerabilities and different types of entry points and check points. Consequently, the grey-box test architecture, the test campaign model, the PenTest workbench, and especially the content of the web security knowledgebase would need to be amended and enhanced to adapt to the penetration testing for the web applications built on more complex architecture.

6.6.4 Simple Web Security Knowledgebase

The proposed test framework includes a simple web security knowledgebase that was created by us from text-based security knowledgebases available on the Web. This simple web security knowledgebase has very limited types and instances of web application vulnerabilities and related attack vectors and fuzz vectors. The proposed test framework relies on security experts to build up a web security knowledgebase that is more comprehensive and updated (see the assumption 3 in section 4.3). This might lead to a more complex security model.

It is not a focus in this thesis to address how to build such a comprehensive and updated web security knowledgebase. This is significant, because the effectiveness of the testing using our framework is limited by how well the security knowledgebase is populated, e.g. with attack vectors. It remains a significant issue that can be an important future work, as discussed in more detail in section 7.2.1.

6.6.5 Limited Scope of the Case Studies

Two real web application projects, namely AEMS and PAL-IS, were used to evaluate the proposed test framework. While the five major components of the framework – the penetration test methodology, the grey-box test architecture, the web security knowledgebase, the test campaign model, and the knowledge-based PenTest workbench, were all applied in the experimental test campaigns and therefore got validated by the case studies to some degree, the validity of the evaluation is weakened due to the limited scope of the case studies in terms of the size and duration of the projects and the size of the web security knowledgebase.

Based on the case studies, the thesis researcher does not know for sure how effectively average testers can use the framework efficiently since the researcher was always available as a resource. In the case studies, the thesis researcher was there to coordinate and guide the development and test teams through the framework both for the test campaign management and the interactions between developers and testers defined by the methodology. In the AEMS case study the development and test team would not have been able to follow the framework that was used during that case study. But based on the experience the thesis researcher formalized the documents and structured the interaction better including the creation of guidelines on how to handle false negatives etc. Based on the experience with the PALIS development and test team, the thesis researcher feels that a tester could follow the framework to performance penetration testing if a well-written user guide is provided and a 1 – 5 day training course is offered to them.

Eventually, a case study should be done more formally with bigger industrial projects to further validate the approach and remove the thesis researcher's participation as a penetration tester from the case study.

The Size and Duration of the Projects

Both projects have a small size of development team that consists of three to five system analysts and developers and one penetration tester. Although we observed active collaboration occurring between the developers and the tester, and how the security knowledge was retained and transferred within the team, it is insufficient to evaluate the benefits gained in such a small team size. Specifically, it is difficult to measure the improvements on test efficiency in quantity when there is only one penetration tester.

Both projects were developed in less than one year with a couple of alpha and beta releases. Although we observed the benefits by reusing the test artifacts and the retained security expertise during the test maintenance and regression testing when AEMS and PAL-IS were under development, we did not follow up the two applications in their application maintenance phase. It is difficult to measure the improvements on test efficiency in quantity with only a couple of test cycles.

The Complexity of the Web Application Architecture

Both AEMS and PAL-IS are .NET based classic database-driven web application with trivial client-side scripting. The experience with the experimental test campaigns is not sufficient to provide solid indication to the efficiency and effectiveness of the test framework when it is applied to more sophisticated AJAX web applications with rich client-side scripting.

Chapter 7. Conclusions

This thesis is concluded by summarizing the major contributions of the thesis research while highlighting some limitations that point towards future research work.

The proposed test framework is complementary to the existing methodologies, security tools, and best industry practices for web application security assurance. The framework can be integrated into a security-oriented software development life cycle, like the MS SDL, to enhance the penetration testing practice. Security tools, like vulnerability scanner, are still used in the framework, while the framework may help to eliminate false negatives. The framework does not guarantee to produce a vulnerability free, completely secure system. Even if the framework is correctly applied in the development of a web application, it is still very likely that a security expert will find vulnerabilities in the application. Rather the application of the framework in software development can act as a benchmark to ensure that basic security has been achieved. It provides an indicator to a security expert of the quality of development team's work: if there are significant failures in the penetration test results, it demonstrates that the development team probably did not apply all security defense mechanisms appropriately.

7.1. Summary of Contributions

In this thesis, we proposed a model-driven penetration test framework for web applications in the context of a security-oriented software development life cycle. The thesis research has made the following major contributions:

Contribution 1:

We proposed a systematic web application penetration test methodology that is fully integrated into a security-oriented software development life cycle. The test methodology specifies the fundamental roles in a test campaign; the development artifacts utilized and the test artifacts produced; and the interactions between penetration tester and developer. The test methodology was completely followed in the PAL-IS case study. In the PAL-IS test campaign, the development artifacts were utilized directly to develop penetration tests. The failures in the test results were addressed at a timely pace: the real defects (the SQL Injection vulnerability found in the first round of testing) were fixed in the following build, and the false negatives (the XSS vulnerability found in the first round of testing) were eliminated using a more comprehensive check point in the application log as a result of collaboration between the penetration tester and the developers.

The test methodology was developed based on a classic iterative, waterfall development process. It needs to be amended to adapt to different development situations when different software development methodologies are adopted.

Contribution 2:

We proposed a model-driven penetration test framework that was applied in all the three case studies. Some test artifacts, including Potential Vulnerability, Attack, and Test Case, were generated automatically using the PenTest workbench. The test artifacts and development artifacts were saved in the test campaign database and were reused in the test campaigns.

The web security knowledgebase proposed and used in the case studies is simple with limited content. It is one of major limitations in the test framework as well as in the case studies. It needs significant improvements as an important future work.

Contribution 3:

We implemented a prototype Knowledge-based PenTest Workbench, together with a simple Web Security Knowledgebase and a Test Campaign Database, which were used in all the three case studies to support the model-driven test approach.

The prototype PenTest workbench includes a few primitive tools that have to be enhanced when the test framework is applied to a larger scale, more complex case study as a future work.

7.2. Future Work

There are three major directions we think that would be worth to pursue in future work.

7.2.1 Build Up More Sophisticated Web Security Knowledgebase

The web security knowledgebase used in the case studies was created from a simple object model that was developed based on the analysis of limited types of vulnerabilities and related attack vectors and fuzz vectors. The web security knowledgebase is implemented in a relational database. As a future work, we think it is important to build a more sophisticated web security knowledgebase for the test framework which can represent a broader spectrum of vulnerabilities, attack vectors, and fuzz vectors. Specifically, future research could address three issues below:

- Leverage an ontology management system, e.g. Protégé-OWL (Protege Home Page), which is capable of representing and managing sophisticate knowledge system to build up a more advanced web security knowledgebase.
- Evaluate the effort for a security expert to create and maintain such an advanced web security knowledgebase for a single web application.
- Explore if it is possible to create a single generic web security knowledgebase that can be applied to any web application, or at least, security experts can select a subset from such a generic web security knowledgebase for a single web application.

7.2.2 Conduct Case Study in Larger Scope

As the case studies conducted in the thesis research have limited scope which impairs the validity of the evaluation of the proposed test framework, a case study in a larger scope should be pursued so that more systematic, quantified and rigorously conducted test campaigns can be made to fully evaluate the framework. This includes more objective means of evaluating the benefits of the approach. For example, a set of pre-defined questionnaires that used to characterize developers and testers at the beginning of the case study, and questionnaires that used to assess the efficiency and effectiveness at the end of the case study. Specifically, this case study could be used to evaluate the proposed framework against the criteria of Resultant Quality and Return on Investment (ROI), which was discussed in section 4.2 but deliberately excluded in the evaluation of this thesis.

7.2.3 Develop a Model-Drive Test Architecture

The model-driven test approach defined and used in the proposed framework is an informal approach, in that it does not formally define Platform Independent Model (PIM), Platform Specific Model (PSM) and model transformation rules, which are three fundamental components in a formalized Model-Driven Architecture (MDA), for the penetration test framework.

We are not aware of any existing MDA that can be used directly for the proposed test framework. It will be interesting to explore the development of such MDA as a future research work. Essentially, the future work will include:

- Present formally a PIM for penetration testing, which involves the investigation of existing PIM modeling languages for one that is feasible to penetration testing, and construct the PIM using the modeling language. If there is no such PIM modeling language, a new one would need to be created.
- Same work as above for PSM for penetration testing.
- When both the PIM and the PSM are formally defined, transformation knowledge (rules) could be specified using a language like QVT (Query/View/Transformation).
- Implement a prototype of the PIM, the PSM, the transformation knowledge, and a transformer. Then apply the prototype to a case study, and then evaluate the prototype based on the case study.

7.2.4 Extend the Proposed Test Framework

The proposed framework in this thesis deals with the most basic penetration test activities and artifacts at each test phase. There are other security assurance tasks that may provide useful inputs to the framework, e.g. to increase vulnerability coverage that the framework can achieve. Specifically, the output of threat analysis for a web application and the security policies with which the IT environment (where the web application will run) must comply, can be leveraged by the framework as useful inputs to provide additional security test requirements. The framework can be extended by defining a specific task at the test analysis phase to process the inputs from threat analysis and security policies, together with the identified potential vulnerabilities, to form more comprehensive test requirements.

Bibliography

The US DoD Trusted Computer System Evaluation Criteria. (1985, December 26). Retrieved September 27, 2011, from NIST Computer Security Resource Center: http://csrc.nist.gov/publications/history/dod85.pdf

Department of Defense Directive Number 8500.01E. (2002, Octomber 24). Retrieved September 27, 2011, from The US Defense Technical Information Center.

XHTML 1.0 Home Page, Second Edition. (2002, August 1). Retrieved October 10, 2011, from W3C XHTML: http://www.w3.org/TR/xhtml1/

XSS vulnerabilty in ASP.Net. (2005, February 17). Retrieved March 2, 2011, from SecurityFocus BugTraq: http://www.securityfocus.com/archive/1/390751/30/0/threaded

Widescale Unicode Encoding Implementation Flaw Discovered. (2007, May 22). Retrieved March 2, 2011, from CGISecurity: http://www.cgisecurity.com/2007/05/widescale-unico.html

The Common Criteria Part I: Introduction and general model. (2009, July). Retrieved September 27, 2011, from The Common Criteria Portal: http://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R3.pdf

The Common Criteria Part II: Security functional requirements. (2009, July). Retrieved September 27, 2011, from The Common Criteria Portal: http://www.commoncriteriaportal.org/files/ccfiles/CCPART2V3.1R3.pdf

The Common Criteria Part III: Security assurance requirements. (2009, July). Retrieved September 27, 2011, from The Common Criteria Portal: http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R3.pdf

The Common Methodology for Information Technology Security Evaluation. (2009, July). Retrieved September 27, 2011, from The Common Criteria Portal: http://www.commoncriteriaportal.org/files/ccfiles/CEMV3.1R3.pdf

Common Criteria Scheme Guide #3 Evaluation Facility Approval. (2010, October). Retrieved September 28, 2011, from Common Criteria Scheme Documentation: http://www.cse-cst.gc.ca/its-sti/services/cc/documentation-eng.html

Microsoft Security Development Lifecycle Version 5.0. (2010, March 31). Retrieved March 2, 2011, from Microsoft Security Development Lifecycle: http://www.microsoft.com/downloads/en/confirmation.aspx?FamilyID=7d8e6144-8276-4a62-a4c8-7af77c06b7ac&displaylang=en

Simplified Implementation of the Microsoft SDL. (2010, Feburary 2). Retrieved March 2, 2011, from Microsoft Security Development Lifecycle: http://www.microsoft.com/downloads/en/confirmation.aspx?FamilyID=0baff8e8-ab17-4e82-a1ff-7bf8d709d9fb&displaylang=en

About the SDL Process. (2011, March 2). Retrieved March 2, 2011, from Microsoft Security Development Lifecycle: http://www.microsoft.com/security/sdl/about/process.aspx

Apache HTTPClient Home Page. (2011, March 2). Retrieved March 2, 2011, from Apache HTTPClient: http://hc.apache.org/index.html

Bugtraq Home Page. (2011, Maarch 2). Retrieved March 2, 2011, from SecurityFocus BugTraq: http://www.securityfocus.com/archive/1

Eclipse Home Page. (2011, March 2). Retrieved March 2, 2011, from Eclipse: http://www.eclipse.org/

HTML 5 Home Page, Draft Version. (2011, May 25). Retrieved October 10, 2011, from W3C HTML 5: http://www.w3.org/TR/html5/

ISECOM OSSTMM Home Page. (2011, March 2). Retrieved March 2, 2011, from ISECOM Open Source Security Testing Methodology Manual: http://www.isecom.org/osstmm/

ISO Freely Available Standards. (2011, September). Retrieved September 27, 2011, from ISO Standards Maintenance Portal: http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

Java SE 6. (2011, March 2). Retrieved March 2, 2011, from Java SE 6 Home Page: http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136246.html

Microsoft Security and Protection: SQL Injection. (2011, September 21). Retrieved Septemeber 21, 2011, from MSDN Library: http://msdn.microsoft.com/en-us/library/ms161953.aspx

Microsoft Security Development Lifecycle: Risks and Impacts of Computer Crime. (2011, March 2). Retrieved March 2, 2011, from Microsoft Security Development Lifecycle: http://www.microsoft.com/security/sdl/about/whysdl.aspx

MITRE CVE Home Page. (2011, March 2). Retrieved March 2, 2011, from Mitre Common Vulnerabilities and Exposures: http://cve.mitre.org/

MITRE CWE Home Page. (2011, March 2). Retrieved March 2, 2011, from Mitre Common Weakness Enumeration: http://cwe.mitre.org/

Model Driven Architecture Home Page. (2011, March 2). Retrieved March 2, 2011, from OMG Model Driven Architecture: http://www.omg.org/mda/

mybatis Home Page. (2011, March 2). Retrieved March 2, 2011, from ibaitis: http://www.mybatis.org/

MySQL Home Page. (2011, March 2). Retrieved March 2, 2011, from MySQL: http://www.mysql.com/

Nessus Home Page. (2011, March 2). Retrieved March 2, 2011, from Nessus: http://nessus.org/products/professional-feed/?gclid=CPrsmpqct6UCFRLLKgodXHcGXw

Nikto Home Page. (2011, March 2). Retrieved March 2, 2011, from Nikto: http://www.cirt.net/nikto2

Nmap Home Page. (2011, March 2). Retrieved March 2, 2011, from Nmap: http://nmap.org/

NVD Home Page. (2011, March 2). Retrieved March 2, 2011, from National Vulnerability Database Version 2.2: http://nvd.nist.gov/

OWASP Cross-site Scripting. (2011, March 2). Retrieved March 2, 2011, from OWASP Cross-site Scripting: http://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

OWASP DOM Based XSS. (2011, March 2). Retrieved March 2, 2011, from OWASP DOM Based XSS: http://www.owasp.org/index.php/DOM_Based_XSS

OWASP WebGoat Project. (2011, October 2). Retrieved October 2, 2011, from The Open Web Application Security Project: https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

OWASP WebGoat User and Install Guide. (2011, October). Retrieved October 2, 2011, from The Open Web Application Security Project: https://www.owasp.org/index.php/WebGoat_User_and_Install_Guide_Table_of_ Contents

SQuirreL SQL Client Home Page. (2011, March 2). Retrieved March 2, 2011, from SQuirreL SQL Client: http://squirrel-sql.sourceforge.net/

The Common Criteria Portal Home Page. (2011, September). Retrieved September 28, 2011, from The Common Criteria Portal: http://www.commoncriteriaportal.org/

The OSVDB Hoem Page. (2011, March 2). Retrieved March 2, 2011, from The Open Source Vulnerability Database: http://osvdb.org

US-Cert Vulnerability Notes Database Home Page. (2011, March 2). Retrieved March 2, 2011, from Unite State Computer Emergency Readiness Team: http://www.kb.cert.org/vuls

Allen, J. H., Barnum, S., Ellison, R. J., McGraw, G., & Mead, N. R. (2008). *Software Security Engineering: A Guide for Project Managers.* Addison Wesley.

Andreu, A. (2006). *Professional Pen Testing for Web Applications.* Wrox Press.

Arkin, B., Stender, S., & McGraw, G. (2005, January-February). Software Penetration Testing. *IEEE Security & Privacy, Volume 3*(Issue 1), pp. 84-87.

Arlow, J., & Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design* (Second ed.). Addison Wesley.

Baker, P., Dai, Z. R., Grabowski, J., Haugen, Ø., Schieferdecker, I., & Williams, C. (2008). *Model-Driven Testing Using the UML Testing Profile.* Springer.

Behnam, S. A., Amyot, D., Forster, A. J., Peyton, L., & Shamsaei, A. (May, 2009). Goal-Driven Development of a Patient Surveillance Application for Improving Patient

Safety. *4th International MCeTech Conference on eTechnologies. LNBIP Volume 26*, pp. 65 - 76. Ottawa, Canada: Springer.

Bell, D., Cesare, S. d., Iacovelli, N., Lycett, M., & Merico, A. (2007). A framework for deriving semantic web services. *Information Systems Frontiers, 9*(1), 69-84.

Berners-Lee, T., Fielding, R. T., & Masinter, L. (2005, January). *Uniform Resource Identifier (URI): Generic Syntax.* Retrieved March 2, 2011, from IETF RFC 3986: http://www.ietf.org/rfc/rfc3986.txt

Berners-Lee, T., Masinter, L., & McCahill, M. (1994, December). *Uniform Resource Locators (URL).* Retrieved March 2, 2011, from IETF RFC 1738: http://www.ietf.org/rfc/rfc1738.txt

Bialkowski, J., & Heineiman, K. (Eds.). (2004, May). *Application Vulnerability Description Language v1.0.* Retrieved March 2, 2011, from http://www.oasis-open.org/committees/download.php/7145/AVDL Specification V1.pdf

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional.

Bishop, M. (2007, November-December). About Penetration Testing. *IEEE Security & Privacy, Volume 5*(Issue 6), pp. 84-87.

Burnstein, I. (2003). *Practical Software Testing: A Process-Oriented Approach.* Springer-Verlag.

Cascading Style Sheets home page. (n.d.). Retrieved 12 31, 2011, from W3C Cascading Style Sheets Web Site: http://www.w3.org/Style/CSS/Overview.en.html

Common Criteria Schema Overview. (n.d.). Retrieved September 28, 2011, from Canadian Common Criteria Schema: http://www.cse-cst.gc.ca/its-sti/services/cc/ccso-vesccc-eng.html

Dai, Z. R. (2004). Model-Driven Testing with UML 2.0. *The Second European Workshop on Model Driven Architecture.*

Daw, M. (2009). *Input Validation Cheat Sheet.* Retrieved September 2009, from michaeldaw.org: http://michaeldaw.org/input_validation_cheat_sheet

Daw, M. (2009). *SQL Injection Cheat Sheet.* Retrieved September 2009, from michaeldaw.org: http://michaeldaw.org/sql-injection-cheat-sheet

Fielding, R. T., Gettys, J., Mogul, J. C., Nielsen, H. F., Masinter, L., Leach, P. J., et al. (1999, June). *Hypertext Transfer Protocol -- HTTP/1.1.* Retrieved March 2, 2011, from IETF RFC 2616: http://www.ietf.org/rfc/rfc2616

Frankel, D. S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley Publishing.

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P. L., Sink, E., et al. (1999, June). *HTTP Authentication: Basic and Digest Access Authentication.* Retrieved March 2, 2011, from IETF RFC 2617 Home Page: http://www.ietf.org/rfc/rfc2617

Galin, D. (2004). *Software Quality Assurance: From theory to implementation.* Pearson Education Limited.

Gaševic, D., Djuric, D., & Devedzic, V. (2006). *Model Driven Architecture and Ontology Development.* Springer.

Grossman, J., Hansen, R., Petkov, P. D., Rager, A., & Fogie, S. (2007). *XSS Attacks: Cross Site Scripting Exploits and Defense.* (S. Fogie, Ed.) Syngress Publishing.

Hansen, R. (2011, March 2). *XSS (Cross Site Scripting) Cheat Sheet.* Retrieved March 2, 2011, from ha.ckers.org: http://ha.ckers.org/xss.html

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly, 28*(1), 75-105.

Hoffman, B., & Sullivan, B. (2008). *Ajax Security.* Addison Wesley.

Hollar, R., & Murphy, R. (2006). *Enterprise Web Service Security.* Charles River Media.

Howard, M., & Lipner, S. (2006). *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software.* Microsoft Press.

Jacky, J., Veanes, M., Campbell, C., & Schulte, W. (2008). *Model-Based Software Testing and Analysis with C#.* Cambridge University Press.

Javed, A. Z., Strooper, P. A., & Watson, G. N. (2007). Automated Generation of Test Cases Using Model-Driven Architecture. *The Second International Workshop on Automation of Software Test.* IEEE Computer Society.

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture Practice and Promise.* Addison-Wesley.

Korpela, J. K. (2006). *Unicode Explained.* O'Reilly.

Kristol, D. M., & Montulli, L. (2000, October). *HTTP State Management Mechanism.* Retrieved March 2, 2011, from IETF RFC 2965 Home Page: http://www.ietf.org/rfc/rfc2965

Manzuik, S., Gold, A., & Gatford, C. (2007). *Network Security Assessment: From Vulnerability to Patch.* Syngress Publishing.

McGraw, G. (2006). *Software Security: Building Security In.* Addison Wesley.

Meier, J., Mackman, A., Vasireddy, S., Dunner, M., Escamilla, R., & Murukan, A. (2003). *Improving Web Application Security: Threats and Countermeasures.* Microsoft Press.

Mellor, S. J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA Distilled: Principles of Model-Driven Architecture.* Addison Wesley.

MITRE CAPEC. (2011, March 2). Retrieved March 2, 2011, from Common Attack Pattern Enumeration and Classification: http://capec.mitre.org/

Myers, G. J. (2004). *The Art of Software Testing* (Second ed.). John Wiley & Sons.

OVAL, M. (2011, March 2). *MITRE OVAL Home Page*. Retrieved March 2, 2011, from Mitre Open Vulnerability and Assessment Language: http://oval.mitre.org/index.html

OWASP Testing Guide. (2008, December). *OWASP Testing Guide.* Retrieved March 2, 2011, from The Open Web Application Security Project: https://www.owasp.org/images/8/89/OWASP_Testing_Guide_V3.pdf

OWASP TOP 10. (2007). *OWASP TOP 10: The Ten Most Critical Web Application Security Vulnerabilities.* Retrieved March 2, 2011, from The Open Web Application Security Project: http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf

OWASP WebScarab. (2011, March 2). *OWASP WebScarab Project*. Retrieved March 2, 2011, from The Open Web Application Security Project: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

Palmer, S. (2007). *Web Application Vulnerabilities: Detect, Exploit, Prevent.* Syngress Publishing.

Patton, R. (2000). *Software Testing.* Sams .

Pezz, M., & Young, M. (2008). *Software Testing and Analysis: Process, Principles and Techniques.* John Wiley & Sons .

Potter, B., & McGraw, G. (2004, September-October). Software Security Testing. *IEEE Security & Privacy, Volume 2*(Issue 5), pp. 81-85.

Protege Home Page. (n.d.). Retrieved March 15, 2011, from Protege Project: http://protege.stanford.edu/

Raggett, D., Hors, A. L., & Jacobs, I. (1999, December 24). *HTML 4.01 Specification.* Retrieved March 2, 2011, from HTML 4.01 Specification Home Page: http://www.w3.org/TR/html401/

Rescorla, E. (2000, May). *HTTP Over TLS.* Retrieved March 2, 2011, from IETF RFC 2818 Home Page: http://www.ietf.org/rfc/rfc2818

SANS TOP 20. (2011, March 2). *TOP 20 Cyber Security Risk*. Retrieved March 2, 2011, from The SANS (SysAdmin, Audit, Network, Security) Institute: http://www.sans.org/top-cyber-security-risks/?ref=top20

SANS TOP 25. (2009, July 27). *2009 CWE/SANS TOP 25 Most Dangerous Programming Errors.* Retrieved March 2, 2011, from Common Weaknesses Enumeration: http://cwe.mitre.org/top25/pdf/2009_cwe_sans_top_25.pdf

Scambray, J., Shema, M., & Sima, C. (2006). *Hacking Exposed Web Applications* (Second ed.). McGraw-Hill.

Schmidt, D. C. (2006, February). Model-Driven Engineering. *IEEE Computer*, pp. 25-32.

Shah, S. (2008). *Web 2.0 Security - Defending AJAX, RIA, and SOA.* Charles River Media.

Shklar, L., & Rosen, R. (2009). *Web Application Architecture: Principles, Protocols and Practices* (Second ed.). John Wiley & Sons.

Singh, I., Stearns, B., Johnson, M., & Team, E. (2002). *Designing Enterprise Applications with the J2EE Platform* (Second ed.). Addison Wesley.

Splaine, S. (2002). *Testing Web Security: Assessing the Security of Web Sites and Applications.* John Wiley & Sons .

Stahl, T., & Voelter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management.* Wiley Publishing.

Stuttard, D., & Pinto, M. (2008). *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws.* John Wiley & Sons.

Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery.* Addison Wesley.

The Common Criteria Introduction. (n.d.). Retrieved September 27, 2011, from The Common Criteria Evaluation and Validation Scheme: http://www.niap-ccevs.org/cc_docs/cc_introduction-v2.pdf

Thomas, H., & Chase, S. (2005). *The Software Vulnerability Guide.* Cengage Charles River Media.

Thompson, H. (2005, January-February). Application Penetration Testing. *IEEE Security & Privacy, Volume 3*(Issue 1), pp. 66-69.

WASC. (2010, January). *The WASC Threat Classification v2.0.* Retrieved March 2, 2011, from Web Application Security Consortium: http://projects.webappsec.org/f/WASC-TC-v2_0.pdf

Wells, C. (2007). *Securing Ajax Applications.* O'Reilly.

Wysopal, C., Nelson, L., Zovi, D. D., & Dustin, E. (2007). *The Art of Software Security Testing: Identifying Software Security Flaws.* Addison Wesley.

Appendix A1 WebGoat Test Analysis Document[4]

Document: Phase I Penetration Test Analysis
Project: WebGoat
Author: Pulei Xiong
Date: October, 2009

1. Purpose:

Identify potential vulnerabilities and attack vectors

2. Development Artifacts

WebGoat release (version 5.2)

3. Application Footprint

Platform: J2EE 1.4, JRE 1.6.0_01
Components: -
Database: Derby 10.2.1.6
Operating System: Windows XP
Web Server: Apache Tomcat 5.5
Encoding: UTF8

4. Potential Vulnerabilities with related application Functionalities

WebGoat is a deliberately insecure web application. In this case study, four functionalities were selected which have known vulnerabilities of SQL Injection or Cross-Site scripting.

Functionality	**Vulnerability**	**Attack Vector**
Search for weather information by city name	Standard SQL Injection	To bypass SQL query criteria to get weather of all cities
Search for a user	Standard SQL Injection	To bypass SQL query criteria to get all users' data
Post a message in message board	Stored Cross-Site Scripting	Insert malicious scripts in the posted message and get other users to view it
Search	Reflected Cross-Site Scripting	Insert malicious scripts in search criteria that is returned to browser and gets executed at the client-side

[4] This document was not created originally in the case study. It was created later on based on the existing test artifacts by applying the test analysis document template for better presentation.

Appendix A2 WebGoat Test Design Document[5]

Document: Phase II Penetration Test Design
Project: WebGoat
Author: Pulei Xiong
Date: October, 2009

1. Purpose:

Develop penetration test cases based on the sample web security knowledgebase to cover all the potential vulnerabilities that have been identified at the test analysis phase (Phase I).

2. Development Artifacts

WebGoat release (version 5.2)

3. Application Footprint

Platform: J2EE 1.4, JRE 1.6.0_01
Components: -
Database: Derby 10.2.1.6
Operating System: Windows XP
Web Server: Apache Tomcat 5.5
Encoding: UTF8

4. Entry Points with Attacks & Check Points

```
--------------- List of Entry Points --------------
# of entry points: 13
# of attacks: 13
```

Entry point: Launch WebGoat Home Page (ID: ep001)

```
GET http://127.0.0.1:80/WebGoat/attack
Functionality: Navigation
```

Entry point: Log on using HTTP Basic Authentication (ID: ep002)

```
POST http://127.0.0.1:80/WebGoat/attack
Form Fields:
   Username=guest
   Password=guest
Functionality: Basic Authentication
```

Entry point: Launch WebGoat Course Page (ID: ep003)

```
POST http://127.0.0.1:80/WebGoat/attack
Form Fields:
  start=Start WebGoat
Functionality: Navigation
```

[5] This document was not created originally in the case study. It was created later on based on the existing test artifacts by applying the test design document template for better presentation.

Entry point: Navigate to the "Phishing with XSS" page (ID: ep004)
GET http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Functionality: Navigate

Entry point: Search Employee (ID: ep005)
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Form Fields:
Username
SUBMIT=Search
Functionality: Search

Vulnerability: Reflected Cross-Site Scripting

Attack: 1
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field:
Username=<script>alert('XSS');</script>

Defense Mechanism: None
Check Point: "Search Employee" Page
Expected result -> the response page is displayed with a popup window with "XSS" message.

Attack: 2
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field:
Username=>"><script>alert('XSS')</script>&

Defense Mechanism: None
Check Point: "Search Employee" Page
Expected result -> the response page is displayed with a popup window with "XSS" message.

Attack: 3
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field:
Username=%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e

Defense Mechanism: None
Check Point: "Search Employee" Page
Expected result -> the response page is displayed with a popup window with "XSS" message.

Entry point: Logout (ID: ep006)
GET http://127.0.0.1:80/WebGoat/attack?action=Logout
Functionality: Logout

Entry point: Navigate to "Stored XSS Attacks" page (ID: ep007)

```
GET http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Functionality: Navigate
```

Entry point: Post a message (ID: ep008)

```
POST http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Form Fields:
   title
   message
   SUBMIT=Submit
Functionality: Post a message
```

Vulnerability: Reflected Cross-Site Scripting

Attack: 4

```
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field:
message=<script>alert('XSS');</script>
```

Check Point: "View the Message" Page

Expected result -> when click on the link of the posted message, the response page is displayed with a popup window with "XSS" message.

Attack: 5

```
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field:
message=>"><script>alert('XSS')</script>&
```

Check Point: "View the Message" Page

Expected result -> when click on the link of the posted message, the response page is displayed with a popup window with "XSS" message.

Attack: 6

```
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field:
message=%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e
```

Defense Mechanism: None
Check Point: "View the Message" Page

Expected result -> when click on the link of the posted message, the response page is displayed with a popup window with "XSS" message.

Attack: 7

```
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
```

Manipulated Form Field: message=<IMG
SRC="javascript:alert('XSS');">

Check Point: “View the Message” Page
Expected result -> when click on the link of the posted message, the response page is displayed with a popup window with “XSS” message.

Attack: 8
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field: message=<IMG
SRC=javascript:alert('XSS')>

Check Point: “View the Message” Page
Expected result -> when click on the link of the posted message, the response page is displayed with a popup window with “XSS” message.

Attack: 9
Attack Description: Reflected XSS attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Manipulated Form Field:
message=<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114
;&#105;&#112;&<WBR>#116;&#58;&#97;&#108;&#101;&<WBR>#114;&#
116;&#40;&#39;&#88;&#83<WBR>;&#83;&#39;&#41>

Defense Mechanism: None
Check Point: “View the Message” Page
Expected result -> when click on the link of the posted message, the response page is displayed with a popup window with “XSS” message.

<u>Entry point: View the message (ID: ep009)</u>
GET http://127.0.0.1:80/WebGoat/attack?Num=1
Functionality: View a message

<u>Entry point: Navigate to the “Numeric SQL Injection” page (ID: ep010)</u>
GET http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
Functionality: Navigate

<u>Entry point: Retrieve City Weather (ID: ep011)</u>
POST http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
Form Fields:
Station
SUBMIT=Go!
Functionality: Retrieve Data

Vulnerability: Reflected Cross-Site Scripting

Attack: 10

Attack Description: Standard SQL Injection attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
Manipulated Form Field: Station=1 or 1=1--

Check Point: "Retrieve City Weather" Page
Expected result -> the response page is displayed with the weather information for all cities.

Attack: 11
Attack Description: Standard SQL Injection attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
Manipulated Form Field: Station=a' or 'a'='a

Check Point: "Retrieve City Weather" Page
Expected result -> the response page is displayed with error message "Error parsing station as a number: For input string: a' or 'a'='a".

Entry point: Navigate to the "String SQL Injection" page (ID: ep012)
GET http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
Functionality: Navigate

Entry point: Retrieve Employee Profile (ID: ep013)
POST http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
Form Fields:
account_name
SUBMIT=Go!
Functionality: Retrieve Data

Vulnerability: Reflected Cross-Site Scripting

Attack: 12
Attack Description: Standard SQL Injection attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
Manipulated Form Field: account_name=1 or 1=1--

Check Point: "Retrieve Employee Profile" Page
Expected result -> the response page is displayed with message "No results matched. Try Again.".

Attack: 13
Attack Description: Standard SQL Injection attack
POST http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
Manipulated Form Field: account_name=a' or 'a'='a

Check Point: "Retrieve Employee Profile" Page
Expected result -> the response page is displayed with all employees' information.

5. User Scenarios

--------------- List of User Scenarios --------------
# of user scenarios: 4

__**User Scenario 1: Phishing with XSS**__
Step1: Entry Point: Launch WebGoat Home Page (ID: ep001)
Description: In Browser, enter URL "http://127.0.0.1:80/WebGoat/attack" to open login page.

Step2: Entry Point: Log on using HTTP Basic Authentication (ID: ep002)
Description: Enter "guest" as username, "guest" as password, and click on "Log In" button to log on WebGoat.

Step3: Entry Point: Launch WebGoat Course Page (ID: ep003)
Description: Click on "Start WebGoat" button to launch WebGoat course page.

Step4: Entry Point: Navigate to the "Phishing with XSS" page (ID: ep004)
Description: Click on "Cross-Site Scripting (XSS)" navigation menu at the left of the web page to expand it, and then click on "Phishing with XSS" link to launch the page.

Step5: Entry Point: Search Employee (ID: ep005)
Description: Enter "Tom" in the search text field, and then click on "Search" button.

Step6: Entry Point: Logout (ID: ep006)
Description: Click on "Logout" link to log out WebGoat.

__**User Scenario 2: Stored XSS Attack**__
Step1: Entry Point: Launch WebGoat Home Page (ID: ep001)
Description: In Browser, enter URL "http://127.0.0.1:80/WebGoat/attack" to open login page.

Step2: Entry Point: Log on using HTTP Basic Authentication (ID: ep002)
Description: Enter "guest" as username, "guest" as password, and click on "Log In" button to log on WebGoat.

Step3: Entry Point: Launch WebGoat Course Page (ID: ep003)
Description: Click on "Start WebGoat" button to launch WebGoat course page.

Step4: Entry Point: Navigate to the "Stored XSS Attack" page (ID: ep007)
Description: Click on "Cross-Site Scripting (XSS)" navigation menu at the left of the web page to expand it, and then click on "Stored XSS Attack" link to launch the page.

Step5: Entry Point: Post a message (ID: ep008)
Description: Enter "Title1" in field "title", enter "For test" in field "message", and then click on "Submit" button.

Step6: Entry Point: View the message (ID: ep009)
Description: Click on the link to the title of the posted message to view the message.

Step7: Entry Point: Logout (ID: ep006)
Description: Click on "Logout" link to log out WebGoat.

<u>User Scenario 3: Numeric SQL Injection</u>

Step1: Entry Point: Launch WebGoat Home Page (ID: ep001)
Description: In Browser, enter URL "http://127.0.0.1:80/WebGoat/attack" to open login page.

Step2: Entry Point: Log on using HTTP Basic Authentication (ID: ep002)
Description: Enter "guest" as username, "guest" as password, and click on "Log In" button to log on WebGoat.

Step3: Entry Point: Launch WebGoat Course Page (ID: ep003)
Description: Click on "Start WebGoat" button to launch WebGoat course page.

Step4: Entry Point: Navigate to the "Numeric SQL Injection" page (ID: ep010)
Description: Click on "Injection flaw" navigation menu at the left of the web page to expand it, and then click on "Numeric SQL Injection" link to launch the page.

Step5: Entry Point: Retrieve City Weather (ID: ep011)
Description: Select "Seattle" in the local weather station dropdown list, click on the "Go!" button to view the weather of the selected city.

Step6: Entry Point: Logout (ID: ep006)
Description: Click on "Logout" link to log out WebGoat.

<u>User Scenario 4: String SQL Injection</u>

Step1: Entry Point: Launch WebGoat Home Page (ID: ep001)
Description: In Browser, enter URL "http://127.0.0.1:80/WebGoat/attack" to open login page.

Step2: Entry Point: Log on using HTTP Basic Authentication (ID: ep002)
Description: Enter "guest" as username, "guest" as password, and click on "Log In" button to log on WebGoat.

Step3: Entry Point: Launch WebGoat Course Page (ID: ep003)
Description: Click on "Start WebGoat" button to launch WebGoat course page.

Step4: Entry Point: Navigate to the "String SQL Injection" page (ID: ep012)
Description: Click on "Injection flaw" navigation menu at the left of the web page to expand it, and then click on "String SQL Injection" link to launch the page.

Step5: Entry Point: Retrieve Employee Profile (ID: ep013)
Description: Enter "Snow" in the last name input field, and then click on the "Go!" button to view the user profile.

Step6: Entry Point: Logout (ID: ep006)

```
Description: Click on “Logout” link to log out WebGoat.

--------------------------------------------------
```

6. Test Cases

```
--------------- List of Test Cases --------------
# of test cases: 13
```

Test Case: 1

```
Scenario 1: Phishing with XSS
Step 1
GET URL: “http://127.0.0.1:80/WebGoat/attack”

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900

Step 5: Attack 1
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Input Form Fields:
     Username=<script>alert('XSS');</script>
     SUBMIT=Search

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 2

```
Scenario 1: Phishing with XSS
Step 1
GET URL: “http://127.0.0.1:80/WebGoat/attack”

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
```

```
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900

Step 5: Attack 2
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Input Form Fields:
     Username=>"><script>alert('XSS')</script>&
     SUBMIT=Search

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 3

```
Scenario 1: Phishing with XSS
Step 1
GET URL: "http://127.0.0.1:80/WebGoat/attack"

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900

Step 5: Attack 3
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=13&menu=900
Input Form Fields:
     Username=%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e
     SUBMIT=Search

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 4

```
Scenario 2: Stored XSS Attack
Step 1
GET URL: "http://127.0.0.1:80/WebGoat/attack"

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat
```

```
Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
```

Step 5: **Attack 4**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Input Form Fields:
     title=Title1
     message=<script>alert('XSS');</script>
     SUBMIT=Submit

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?Num=1

Step 7
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 5

```
Scenario 2: Stored XSS Attack
Step 1
GET URL: “http://127.0.0.1:80/WebGoat/attack”

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
```

Step 5: **Attack 5**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Input Form Fields:
     title=Title1
     message=>"><script>alert('XSS')</script>&
     SUBMIT=Submit

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?Num=1

Step 7
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 6

```
Scenario 2: Stored XSS Attack
Step 1
GET URL: “http://127.0.0.1:80/WebGoat/attack”
```

```
Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
```

Step 5: **Attack 6**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Input Form Fields:
     title=Title1
     message=%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e
     SUBMIT=Submit

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?Num=1

Step 7
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 7

```
Scenario 2: Stored XSS Attack
Step 1
GET URL: "http://127.0.0.1:80/WebGoat/attack"

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
```

Step 5: **Attack 7**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Input Form Fields:
     title=Title1
     message=<IMG SRC="javascript:alert('XSS');">
     SUBMIT=Submit

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?Num=1
```

```
Step 7
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 8

```
Scenario 2: Stored XSS Attack
Step 1
GET URL: “http://127.0.0.1:80/WebGoat/attack”

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
```

Step 5: **Attack 8**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Input Form Fields:
     title=Title1
     message=<IMG SRC=javascript:alert('XSS')>
     SUBMIT=Submit

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?Num=1

Step 7
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 9

```
Scenario 2: Stored XSS Attack
Step 1
GET URL: “http://127.0.0.1:80/WebGoat/attack”

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
```

Step 5: **Attack 9**
```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=50&menu=900
Input Form Fields:
     title=Title1
     message=<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114;&#105;&
  #112;&<WBR>#116;&#58;&#97;&#108;&#101;&<WBR>#114;&#116;&#40;&#39;&#88
  ;&#83<WBR>;&#83;&#39;&#41>
     SUBMIT=Submit

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?Num=1

Step 7
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 10
```
Scenario 3: Numeric SQL Injection
Step 1
GET URL: "http://127.0.0.1:80/WebGoat/attack"

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
```

Step 5: **Attack 10**
```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
Form Fields:
     Station=1 or 1=1--
     SUBMIT=Go!

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 11
```
Scenario 3: Numeric SQL Injection
Step 1
GET URL: "http://127.0.0.1:80/WebGoat/attack"

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest
```

```
Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
```

Step 5: **Attack 11**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=67&menu=1200
Form Fields:
     Station=a' or 'a'='a
     SUBMIT=Go!

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 12

```
Scenario 4: String SQL Injection
Step 1
GET URL: "http://127.0.0.1:80/WebGoat/attack"

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
```

Step 5: **Attack 12**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
Input Form Fields:
     account_name=1 or 1=1--
     SUBMIT=Go!

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

Test Case: 13

```
Scenario 4: String SQL Injection
Step 1
GET URL: "http://127.0.0.1:80/WebGoat/attack"

Step 2
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     Username=guest
     Password=guest

Step 3
POST URL: http://127.0.0.1:80/WebGoat/attack
Input Form Fields:
     start=Start WebGoat

Step 4
GET URL: http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
```

Step 5: **Attack 13**

```
POST URL: http://127.0.0.1:80/WebGoat/attack?Screen=58&menu=1200
Input Form Fields:
     account_name=a' or 'a'='a
     SUBMIT=Go!

Step 6
GET URL: http://127.0.0.1:80/WebGoat/attack?action=Logout
```

--

Appendix A3 WebGoat Test Execution Document

Document: Phase III Penetration Test Execution
Project: WebGoat
Author: Pulei Xiong
Date: October, 2009

1. Purpose:

Execute the test cases created at the design phase (Phase II), and then analyze test results and create a test report.

2. Test Result Summary

2.1 Application under Test

WebGoat release, version 5.2

2.2 Test Report

```
------------- Test Campaign Summary --------------
# of identified vulnerabilities: 3
# of vulnerable entry points: 4
# of entry points tested: 13
# of test cases executed: 13
--------------------------------------------------

------- Web Security Repository Statistics -------
# of vulnerabilities: 5
# of attack vectors: 9
# of fuzz vectors: 26
--------------------------------------------------

------- Identified Vulnerability Details -------

Vulnerability: Reflected XSS
-->Entry Point: ep005 (Phishing with XSS)

Vulnerability: Stored XSS
-->Entry Point: ep008 (Stored XSS Attacks)

Vulnerability: Standard SQL Injection
-->Entry Point: ep011 (Numeric SQL Injection)
-->Entry Point: ep013 (String SQL Injection)
--------------------------------------------------
```

Appendix B1 AEMS Test Analysis Document[6]

Document: Phase I Penetration Test Analysis
Project: AEMS
Author: Pulei Xiong
Date: January, 2010

1. Purpose:
Identify potential vulnerabilities and attack vectors

2. Development Artifacts
Software Prototype Implementation (Beta 1)

3. Application Footprint
Platform: .NET Framework (.NET ASP) 3.5 SP1
Components: -
Database: SQL Server 2005
Operating System: Windows 2008 Server
Web Server: IIS 7.0
Encoding: *

4. Potential Vulnerabilities with related application Functionalities
The following table was created by walking through the functionalities of the software prototype implementation (Beta 1), picking up a subset of the functionalities, and evaluating all relevant vulnerabilities described in WebGoat. The functionality column is the link between the two. Vulnerabilities and Attack Vectors are defined in WebGoat. Notes contain the concise descriptions of the functionalities.

Functionality	Vulnerability	Attack Vector	Description
Login	Standard SQL Injection	Attack against password-based logon to bypass authentication	User login
	Password strength	Online & offline attack	
	Insecure communication	Intercept HTTP traffic to steal login credentials	
	DoS	Consecutive login failure to lock a legitimate user	
Search Patient	Standard SQL Injection	Attack against various "search" to bypass role-based authorization	Search a patient or patients
		Attack against various "search" to damage the database	Search a patient or patients
Form Data Collection	Stored Cross-Site Scripting	Get users to view/edit an existing form containing with malicious scripts	Form-based data view and edit to add patients

[6] This document was not created originally in the case study. It was created later on based on the existing test artifacts by applying the test analysis document template for better presentation.

Appendix B2 AEMS Test Design Documents[7]

Document: Phase II Penetration Test Design
Project: AEMS
Author: Pulei Xiong
Date: January, 2010

1. Purpose:

Develop penetration test cases based on the sample web security knowledgebase to cover all the potential vulnerabilities that have been identified at the test analysis phase (Phase I).

2. Development Artifacts

Software Prototype Implementation (Beta 1)

3. Application Footprint

Platform: .NET Framework (.NET ASP) 3.5 SP1
Components: ibatis
Database: SQL Server 2005
Operating System: Windows 2008 Server
Web Server: IIS 7.0
Encoding: *

4. Entry Points with Attacks & Check Points

```
--------------- List of Entry Points --------------
# of entry points: 6
# of attacks: 10
```

Entry point: Launch Login Page (ID: ep001)

```
GET HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Functionality: Navigation
```

Entry point: Log On (ID: ep002)

```
POST HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Form Fields:
   username
   password
   LoginButton=Log In
Functionality: Log On
```

Vulnerability: Standard SQL Injection

Attack: 1

```
Attack Description: Standard SQL Injection attack
```

[7] This document was not created originally in the case study. It was created later on based on the existing test artifacts by applying the test design document template for better presentation.

POST
HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2
f
Manipulated Form Field: password=1 or 1=1--

Defense Mechanism: ibatis component
Check Point: "Login" Page
Expected result -> the Login web page is displayed with error message: *Your login attempt was not successful. Please try again.*

Attack: 2
Attack Description: Standard SQL Injection attack
POST
HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2
f
Manipulated Form Field: password=a' or 'a'='a

Defense Mechanism: ibatis component
Check Point: "Login" Page
Expected result -> the Login web page is displayed with error message: *Your login attempt was not successful. Please try again.*

Entry point: Log Out (ID: ep003)
GET HTTPS://peilos.servebeer.com:443/Account/Logout
Functionality: Log Out

Entry point: Add A Patient (ID: ep004)
POST HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Form Fields:
```
   Pid=-1
   RecordNumber
   FirstName
   LastName
   DateOfBirth
   IllnessHistory
   Gender
   Location
   Diagnosis
   PhysicianName
   CTUDate
   CTUTime
```
Functionality: Edit Profile

Vulnerability: Stored Cross-Site Scripting

Attack: 3
Attack Description: Stored XSS attack
POST
HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Manipulated Form Field:
IllnessHistory=<script>alert('XSS');</script>

Defense Mechanism: ASP.NET
Check Point: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*

Attack: 4
Attack Description: Stored XSS attack
POST
HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Manipulated Form Field:
```
IllnessHistory=>"><script>alert('XSS')</script>&
```

Defense Mechanism: ASP.NET
Check Point: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*

Attack: 5
Attack Description: Stored XSS attack
POST
HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Manipulated Form Field:
```
IllnessHistory=%uff1cscript%uff1ealert('XSS')%uff1c/script%
uff1e
```

Defense Mechanism: ASP.NET
Check Point: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*

Attack: 6
Attack Description: Stored XSS attack
POST
HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Manipulated Form Field:
```
IllnessHistory=<IMG SRC="javascript:alert('XSS');">
```

Defense Mechanism: ASP.NET
Check Point: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*

Attack: 7

Attack Description: Stored XSS attack
POST
HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Manipulated Form Field:
IllnessHistory=<IMG SRC=javascript:alert('XSS')>

Defense Mechanism: ASP.NET
Check Point: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*

Attack: 8
Attack Description: Stored XSS attack
POST
HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Manipulated Form Field:
IllnessHistory=<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114;&#105;&#112;&<WBR>#116;&#58;&#97;&#108;&#101;&<WBR>#114;&#116;&#40;&#39;&#88;&#83<WBR>;&#83;&#39;&#41>

Defense Mechanism: ASP.NET
Check Point: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*

<u>Entry point: Navigate to Patient List Page (ID: ep005)</u>
GET HTTPS://peilos.servebeer.com:443/Observer/PatientList
Functionality: Navigate

<u>Entry point: Search A Patient (ID: ep006)</u>
GET
HTTPS://peilos.servebeer.com:443/Observer/SearchPatient/search=First Name&value=Smith&OK=
Functionality: Retrieve Data

Vulnerability: Standard SQL Injection

Attack: 9
Attack Description: Standard SQL Injection attack
GET
HTTPS://peilos.servebeer.com:443/Observer/SearchPatient/search=First Name&value=Smith&OK=
Manipulated Form Field: value=1 or 1=1--

Defense Mechanism: ibatis component
Check Point: "Search Patient" Page
Expected result -> the Search Patient web page is displayed with none patient, and with error message: *the search text cannot contain any spaces*

Attack: 10
Attack Description: Standard SQL Injection attack
GET
HTTPS://peilos.servebeer.com:443/Observer/SearchPatient/sea
rch=First Name&value=Smith&OK=
Manipulated Form Field: value=a' or 'a'='a

Defense Mechanism: ibatis component
Check Point: "Search Patient" Page
Expected result -> the Search Patient web page is
displayed with none patient

----

5. User Scenarios

--------------- List of User Scenarios --------------
# of user scenarios: 3

User Scenario 1: Login and then Logout
Step1: Entry Point: Launch Login Page (ID: ep001)
Description: In Browser, enter URL
"HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f" to open
login page

Step2: Entry Point: Log On (ID: ep002)
Description: On the login page, input "pxiong" for field "username",
input "123456" for field "password", and click on "Log In" button to
log in AEMS as user "pxiong"

Step3: Entry Point: Logout (ID: ep003)
Description: Click on "logout" button to log out AEMS

User Scenario 2: Add A Patient
Step1: Entry Point: Launch Login Page (ID: ep001)
Description: In Browser, enter URL
"HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f" to open
login page

Step2: Entry Point: Log On (ID: ep002)
Description: On the login page, input "pxiong" for field "username",
input "123456" for field "password", and click on "Log In" button to
log in AEMS as user "pxiong"

Step3: Add A Patient (ID: ep004)
Description: On the "Add a patient" page, input "patient1" in field
"FirstName", input "test" in field "LastName", input "01/02/2010" in
field "DateOfBirth", input "this is for test" in field "IllnessHistory",
select "Male" in field "Gender", input "Medical Floor" in field
"Location", set "Diagnosis" flag as "1", set "PhysicianName" as "1",
input "04/02/2010" in field "CTUDate", input "9:49 PM" in field
"CTUTime"

Step4: Entry Point: Logout (ID: ep003)
Description: Click on "logout" button to log out AEMS

<u>User Scenario 3: Search a patient</u>
Step1: Entry Point: Launch Login Page (ID: ep001)
Description: In Browser, enter URL "HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f" to open login page

Step2: Entry Point: Log On (ID: ep002)
Description: On the login page, input "pxiong" for field "username", input "123456" for field "password", and click on "Log In" button to log in AEMS as user "pxiong"

Step3: Entry Point: Navigate to Patient List Page (ID: ep005)
Description: Click on "Patient List" link to navigate to Patient List page

Step4: Entry Point: Search a Patient (ID: ep006)
Description: On the Patient List page, select "First Name" as "search", input "Smith" in field "value", and click on "Search" button

Step5: Entry Point: Logout (ID: ep003)
Description: Click on "logout" button to log out AEMS

----

6. Test Cases

--------------- List of Test Cases --------------
# of test cases: 10

<u>Test Case: 1</u>
Scenario 1: Login and then Logout
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2: **Attack 1**
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
```
      username=pxiong
      password=1 or 1=1—
      LoginButton=Log In
```

Step 3
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout

<u>Test Case: 2</u>
Scenario 1: Login and then Logout
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

```
Step 2: Attack 2
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=a' or 'a'='a
      LoginButton=Log In

Step 3
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
      LoginButton=Log In
```

Test Case: 3

```
Scenario 2:  Add A Patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In

Step 3: Attack 3
POST HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Input Form Fields:
     Pid=-1
     RecordNumber=
     FirstName=patient2
     LastName=test
     DateOfBirth=01/02/2010
     IllnessHistory=<script>alert('XSS');</script>
     Gender=Male
     Location=Medical Floor
     Diagnosis=1
     PhysicianName=1
     CTUDate=04/02/2010
     CTUTime=9:49 PM

Step 4
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
```

Test Case: 4

```
Scenario 2:  Add A Patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In
```

```
Step 3: Attack 4
POST HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Input Form Fields:
     Pid=-1
     RecordNumber=
     FirstName=patient2
     LastName=test
     DateOfBirth=01/02/2010
     IllnessHistory=>"><script>alert('XSS')</script>&
     Gender=Male
     Location=Medical Floor
     Diagnosis=1
     PhysicianName=1
     CTUDate=04/02/2010
     CTUTime=9:49 PM

Step 4
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
```

Test Case: 5

```
Scenario 2:  Add A Patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In

Step 3: Attack 5
POST HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Input Form Fields:
     Pid=-1
     RecordNumber=
     FirstName=patient2
     LastName=test
     DateOfBirth=01/02/2010
     IllnessHistory=%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e
     Gender=Male
     Location=Medical Floor
     Diagnosis=1
     PhysicianName=1
     CTUDate=04/02/2010
     CTUTime=9:49 PM

Step 4
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
```

Test Case: 6

```
Scenario 2:  Add A Patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
```

```
Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In
```

Step 3: **Attack 6**

```
POST HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Input Form Fields:
     Pid=-1
     RecordNumber=
     FirstName=patient2
     LastName=test
     DateOfBirth=01/02/2010
     IllnessHistory=<IMG SRC="javascript:alert('XSS');">
     Gender=Male
     Location=Medical Floor
     Diagnosis=1
     PhysicianName=1
     CTUDate=04/02/2010
     CTUTime=9:49 PM

Step 4
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
```

Test Case: 7

```
Scenario 2:  Add A Patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In
```

Step 3: **Attack 7**

```
POST HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Input Form Fields:
     Pid=-1
     RecordNumber=
     FirstName=patient2
     LastName=test
     DateOfBirth=01/02/2010
     IllnessHistory=<IMG SRC=javascript:alert('XSS')>
     Gender=Male
     Location=Medical Floor
     Diagnosis=1
     PhysicianName=1
     CTUDate=04/02/2010
     CTUTime=9:49 PM
```

```
Step 4
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
```

Test Case: 8

```
Scenario 2:  Add A Patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In
```

Step 3: **Attack 8**

```
POST HTTPS://peilos.servebeer.com:443/Observer/PersistPatient
Input Form Fields:
     Pid=-1
     RecordNumber=
     FirstName=patient2
     LastName=test
     DateOfBirth=01/02/2010
     IllnessHistory=<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114;
  &#105;&#112;&<WBR>#116;&#58;&#97;&#108;&#101;&<WBR>#114;&#116;&#40;&#
  39;&#88;&#83<WBR>;&#83;&#39;&#41>
     Gender=Male
     Location=Medical Floor
     Diagnosis=1
     PhysicianName=1
     CTUDate=04/02/2010
     CTUTime=9:49 PM

Step 4
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
```

Test Case: 9

```
Scenario 3:  Search a patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In

Step 3
GET URL: HTTPS://peilos.servebeer.com:443/Observer/PatientList
```

Step 4: **Attack 9**

```
GET
HTTPS://peilos.servebeer.com:443/Observer/SearchPatient/search=First
Name&value=1 or 1=1--&OK=

Step 5
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout
```

Test Case: 10

```
Scenario 3:  Search a patient
Step 1
GET URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f

Step 2:
POST URL: HTTPS://peilos.servebeer.com:443/Account/Login?ReturnUrl=%2f
Input Form Fields:
      username=pxiong
      password=123456
      LoginButton=Log In

Step 3
GET URL: HTTPS://peilos.servebeer.com:443/Observer/PatientList
```

Step 4: **Attack 10**

```
GET
HTTPS://peilos.servebeer.com:443/Observer/SearchPatient/search=First
Name&value= a' or 'a'='a &OK=

Step 5
GET URL: HTTPS://peilos.servebeer.com:443/Account/Logout

-------------------------------------------------
```

Appendix B3 AEMS Test Execution Document

Document: Phase III Penetration Test Execution
Project: AEMS
Author: Pulei Xiong
Date: January, 2010

1. Purpose:
Execute the test cases created at the design phase (Phase II), and then analyze test results and create a test report.

2. Test Result Summary (First round)
2.1 Application under Test
AEMS, version Beta 1

2.2 Test Report
```
------------- Test Campaign Summary --------------
# of identified vulnerabilities: 2
# of vulnerable entry points: 2
# of entry points tested: 6
# of test cases executed: 10
--------------------------------------------------

------- Web Security Repository Statistics -------
# of vulnerabilities: 5
# of attack vectors: 9
# of fuzz vectors: 26
--------------------------------------------------

------- Identified Vulnerability Details -------
Vulnerability: Stored XSS
-->Entry Point: ep004 (Add a Patient)

Vulnerability: Standard SQL Injection
-->Entry Point: ep007 (Search Patient Page)
--------------------------------------------------
```

3. Test Result Summary (Regression)
3.1 Application under Test
AEMS, version Beta 2

3.2 Test Report
```
------------- Test Campaign Summary --------------
# of identified vulnerabilities: 1
# of vulnerable entry points: 1
# of entry points tested: 6
# of test cases executed: 10
--------------------------------------------------
```

```
------- Web Security Repository Statistics -------
# of vulnerabilities: 5
# of attack vectors: 9
# of fuzz vectors: 26
--------------------------------------------------

------- Identified Vulnerability Details -------
Vulnerability: Stored XSS
-->Entry Point: ep004 (Add a Patient)
--------------------------------------------------
```

Appendix C1 PAL-IS Test Analysis Document

Document: Phase I Penetration Test Analysis
Project: PAL-IS
Author: Pulei Xiong
Date: September 2, 2010

1. Purpose:

Identify potential vulnerabilities and attack vectors

2. Development Artifacts

PAL-IS Project Specification Document
- Section 4.2 – Functionality

PAL-IS Project Use Cases (for demo)
- Login and then logout
- Search a patient
- Update the current user profile
- Registered Patient
- Unregistered patient
- Education Session

Software Mockup (version 0.1)

3. Application Footprint

Platform: .NET Framework (.NET ASP) 3.5 SP1
Components: -
Database: SQL Server 2005
Operating System: Windows 2008 Server
Web Server: IIS 7.0
Encoding: *

4. Potential Vulnerabilities with related application Functionalities

The following table was created by reviewing functionality described or shown in development artifacts and evaluating all relevant vulnerabilities described in WebGoat. The functionality column is the link between the two. Vulnerabilities and Attack Vectors are defined in WebGoat. Notes contain the concise descriptions of the functionalities.

Functionality	Vulnerability	Attack Vector	Description
Login	Standard SQL Injection	Attack against password-based logon to bypass authentication	User login
	Password strength	Online & offline attack	
	Insecure communication	Intercept HTTP traffic to steal login credentials	
	DoS	Consecutive login failure to lock a legitimate user	
Search Patient	Standard SQL Injection	Attack against various “search” to bypass role-based authorization	Search a patient or patients
		Attack against various “search” to damage the database	Search a patient or patients
Form Data Collection	Stored Cross-Site Scripting	Get users to view/edit an existing form containing with malicious scripts	Various form-based data view and edit
User Profile Management	Stored Cross-Site Scripting	Get users to view/edit an existing user profile containing with malicious scripts	User profile view and edit
Load Attachments	Stored Cross-Site Scripting	Get users to view a (faked) image/pdf etc. file planted with malicious scripts	Load and view attachments
N/A	Resource enumeration	Scan a deployment to find any .bak, .old etc files	N/A
		Scan html pages to see if here is any userID or password embedded	
		Scan a deployment to see if there is a hidden admin page can be exploited	
ESAS Score interface for IVR input. (Web service)	?	IVR server collects scores from patient and calls a web service or REST interface provided by PALIS to enter the ESAS score.	Out of scope for now

Appendix C2 PAL-IS Test Design Document

Document: Phase II Penetration Test Design
Project: PAL-IS
Author: Pulei Xiong
Date: September 4, 2010

1. Purpose:

Develop penetration test cases based on the sample web security knowledgebase to cover all the potential vulnerabilities that have been identified at the test analysis phase (Phase I).

2. Development Artifacts

PAL-IS Project Specification Document
    Section 5 - Architecture
API for URL on PAL-IS
PAL-IS Project Use Cases (for demo)
    Login and then Logout
    Search a patient
    Update the current user profile
    Registered Patient
    Unregistered patient
    Education Session
Software Mockup (version 0.1)

3. Application Footprint

Platform: .NET Framework (.NET ASP) 3.5 SP1
Components: -
Database: SQL Server 2005
Operating System: Windows 2008 Server
Web Server: IIS 7.0
Encoding: *

4. Entry Points with Attacks & Check Points

```
--------------- List of Entry Points --------------
# of entry points: 8
# of attacks: 10
```

Entry Point: Launch Login Page (ID: ep001)

```
GET: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Functionality: Navigation
```

Entry Point: Log On (ID: ep002)

```
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Form Fields:
      username
      password
      LoginButton=Log In
Functionality: Log On
```

Vulnerability: Standard SQL Injection

Attack: 1
Attack Description: Standard SQL Injection attack
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Manipulated Form Field: password=1 or 1=1--

Defense Mechanism: Application Specific
Check Point 1: "Login" Page
Expected result -> the Login web page is displayed with error message: *Your login attempt was not successful. Please try again.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous SQL injection attack was detected from the client on login (lpeyton=1 or 1=1--)*

Attack: 2
Attack Description: Standard SQL Injection attack
POST: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Manipulated Form Field: password=a' or 'a'='a

Defense Mechanism: Application Specific
Check Point 1: "Login" Page
Expected result -> the Login web page is displayed with error message: *Your login attempt was not successful. Please try again.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous SQL injection attack was detected from the client on login (lpeyton='a' or 'a'='a')*

Entry Point: Logout (ID: ep003)
POST: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
Form Fields:
Functionality: Log Out

Entry Point: Navigate to Search Patient Page (ID: ep004)
GET: http://toh7.site.uottawa.ca:80/palis-test/Patients/SearchPatients.aspx
Functionality: Navigation

Entry Point: Search a Patient (ID: ep005)
POST: http://toh7.site.uottawa.ca:80/palis-test/Patients/SearchPatients.aspx
Form Fields:
    SearchType
    SearchText
Functionality: Retrieve Data

Vulnerability: Standard SQL Injection

Attack: 3
Attack Description: Standard SQL Injection attack
POST: http://toh7.site.uottawa.ca:80/palis-test/Patients/SearchPatients.aspx
Manipulated Form Field: SearchText=1 or 1=1--

Defense Mechanism: Application Specific
Check Point 1: "Search Patient" Page
Expected result -> the Search Patient web page is displayed with none patient, and with error message: *the search text cannot contain any spaces*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *the search text cannot contain any spaces; A potentially dangerous SQL injection attack was detected from the client on Search (txtSearchText=1 or 1=1--)*

Attack: 4
Attack Description: Standard SQL Injection attack
POST: http://toh7.site.uottawa.ca:80/palis-test/Patients/SearchPatients.aspx
Manipulated Form Field: SearchText=a' or 'a'='a

Defense Mechanism: Application Specific
Check Point 1: "Search Patient" Page
Expected result -> the Search Patient web page is displayed with none patient, and with error message: *the search text cannot contain any spaces*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *the search text cannot contain any spaces; A potentially dangerous SQL injection attack was detected from the client on Search (txtSearchText='a' or 'a'=a')*

Entry Point: Navigate to My Profile Page (ID: ep006)
GET: http://toh7.site.uottawa.ca:80/palis-test/Teams/MemberProfile.aspx
Functionality: Navigate

Entry Point: Navigate to Edit Profile Page (ID: ep007)
GET: http://toh7.site.uottawa.ca:80/palis-test/Teams/EditProfile.aspx?UserName=lpeyton
Functionality: Navigate

Entry Point: Update Profile (ID: ep008)
POST: http://toh7.site.uottawa.ca:80/palis-test/Teams/EditProfile.aspx?UserName=lpeyton
Form Fields:
FirstName
LastName
Email

```
      Genders
      Approved=on
      Save=Save
Functionality: Edit Profile
```

Vulnerability: Stored Cross-Site Scripting

Attack: 5
Attack Description: Stored XSS attack
POST: http://toh7.site.uottawa.ca:80/palis-test/Teams/EditProfile.aspx?UserName=lpeyton
Manipulated Form Field:
FirstName=<script>alert('XSS');</script>

Defense Mechanism: ASP.NET
Check Point 1: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous Request.Form value was detected from the client (ctl00$ctl00$MainContent$CenterContent$txtFirstName="<script>alert('XSS')...").*

Attack: 6
Attack Description: Stored XSS attack
POST: http://toh7.site.uottawa.ca:80/palis-test/Teams/EditProfile.aspx?UserName=lpeyton
Manipulated Form Field:
FirstName=>"><script>alert('XSS')</script>&

Defense Mechanism: ASP.NET
Check Point 1: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous Request.Form value was detected from the client (ctl00$ctl00$MainContent$CenterContent$txtFirstName="><script>alert('XSS')...").*

Attack: 7
Attack Description: Stored XSS attack
POST: http://toh7.site.uottawa.ca:80/palis-test/Teams/EditProfile.aspx?UserName=lpeyton
Manipulated Form Field:
FirstName=%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e

Defense Mechanism: ASP.NET
Check Point 1: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous Request.Form value was detected from the client (ctl00$ctl00$MainContent$CenterContent$txtFirstName="*=%uff1cscript%uff1ealert(*'XSS'*)%uff1c/script%uff1e*...").*

Attack: 8
Attack Description: Stored XSS attack
POST: http://toh7.site.uottawa.ca:80/palis-test/Teams/EditProfile.aspx?UserName=lpeyton
Manipulated Form Field: FirstName=<IMG SRC="javascript:alert('XSS');">

Defense Mechanism: ASP.NET
Check Point 1: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous Request.Form value was detected from the client (ctl00$ctl00$MainContent$CenterContent$txtFirstName="*=<IMG SRC="javascript:alert(*'XSS'*)*...").*

Attack: 9
Attack Description: Stored XSS attack
POST: http://toh7.site.uottawa.ca:80/palis-test/Teams/EditProfile.aspx?UserName=lpeyton
Manipulated Form Field: FirstName=<IMG SRC=javascript:alert('XSS')>

Defense Mechanism: ASP.NET
Check Point 1: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous Request.Form value was detected from the client (ctl00$ctl00$MainContent$CenterContent$txtFirstName="*=<IMG SRC=javascript:alert(*'XSS'*)>*...").*

```
Attack: 10
Attack Description: Stored XSS attack
POST: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
Manipulated Form Field:
FirstName=<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#1
14;&#105;&#112;&<WBR>#116;&#58;&#97;&#108;&#101;&<WBR>#114;
&#116;&#40;&#39;&#88;&#83<WBR>;&#83;&#39;&#41>
```

Defense Mechanism: ASP.NET
Check Point 1: "System Error" Page
Expected result -> a System Error page is displayed with error message: *A potentially dangerous value was detected in the form submitted. No changes were saved. Please make sure you don't have any HTML tags in the form.*
Check Point 2: Application log
Expected result -> an entry in the application log with error message: *A potentially dangerous Request.Form value was detected from the client (ctl00$ctl00$MainContent$CenterContent$txtFirstName="=<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114;&#105;...").*

----

5. User Scenarios[8]

--------------- List of User Scenarios --------------
# of user scenarios: 3

User Scenario 1: Login and then Logout
Step1: Entry Point: Launch Login Page (ID: ep001)
Description: In Browser, enter URL "http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx" to open login page

Step2: Entry Point: Log On (ID: ep002)
Description: On the login page, input "lpeyton" for field "username", input "lpeyton" for field "password", and click on "Log In" button to log in PAL-IS as user "lpeyton"

Step3: Entry Point: Logout (ID: ep003)
Description: Click on "logout" link to log out PAL-IS

User Scenario 2: Search a patient
Step1: Entry Point: Launch Login Page (ID: ep001)
Description: In Browser, enter URL "http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx" to open login page

Step2: Entry Point: Log On (ID: ep002)

[8] In this case study, the user scenarios were acquired from the development documented "User Scenarios: section 2". The three user scenarios used in the case study are listed here for better reference.

Description: On the login page, input “lpeyton” for field “username”, input “lpeyton” for field “password”, and click on “Log In” button to log in PAL-IS as user “lpeyton”

Step3: Entry Point: Navigate to Search Patient Page (ID: ep004)
Description: Click on “Search Patients” link to navigate to Search Patient page

Step4: Entry Point: Search a Patient (ID: ep005)
Description: On the Search Patients page, select “LastName” as “SearchType”, input “Smith” in field “SearchText”, and click on “Search” button

Step5: Entry Point: Logout (ID: ep003)
Description: Click on “logout” link to log out PAL-IS

User Scenario 3: Update the current user profile
Step1: Entry Point: Launch Login Page (ID: ep001)
Description: In Browser, enter URL “http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx” to open login page

Step2: Entry Point: Log On (ID: ep002)
Description: On the login page, input “lpeyton” for field “username”, input “lpeyton” for field “password”, and click on “Log In” button to log in PAL-IS as user “lpeyton”

Step3: Entry Point: Navigate to My Profile Page (ID: ep006)
Description: Click on “My Profile” link to navigate to My Profile page

Step4: Entry Point: Navigate to Edit Profile Page (ID: ep007)
Description: Click on “Edit this profile” link to navigate to Edit Profile page

Step5: Entry Point: Update Profile (ID: ep008)
Description: On the Edit Profile page, input “Liam” in field “FirstName”, input “Peyton” in field “LastName”, input “lpeyton@site.uOttawa.ca” in field “Email”, and click on “Save” button to submit updated profile

Step6: Entry Point: Logout (ID: ep003)
Description: Click on “logout” link to log out PAL-IS

----

6. Test Cases

--------------- List of Test Cases --------------
# of test cases: 10

Test Case: 1
Scenario 1: Login and then Logout
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

```
Step 2: Attack 1
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=1 or 1=1—
      LoginButton=Log In

Step 3
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 2

```
Scenario 1:  Login and then Logout
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2: Attack 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=a' or 'a'='a
      LoginButton=Log In

Step 3
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 3

```
Scenario 2:  Search a patient
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Patients/SearchPatients.aspx

Step 4: Attack 3
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Patients/SearchPatients.aspx
Input Form Fields:
      SearchType=LastName
      SearchText=1 or 1=1--

Step 5
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 4

```
Scenario 2:  Search a patient
```

```
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Patients/SearchPatients.aspx
```

Step 4: **Attack 4**

```
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Patients/SearchPatients.aspx
Input Form Fields:
      SearchType=LastName
      SearchText=a' or 'a'='a

Step 5
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 5

```
Scenario 3:  Update the current user profile
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/MemberProfile.aspx

Step 4
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
```

Step 5: **Attack 5**

```
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
Input Form Fields:
      FirstName=<script>alert('XSS');</script>
      LastName=Peyton
      Email=lpeyton@site.uOttawa.ca
      Genders=0
      Approved=on
      Save=Save
```

```
Step 6
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 6

```
Scenario 3:  Update the current user profile
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/MemberProfile.aspx

Step 4
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
```

Step 5: **Attack 6**

```
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
Input Form Fields:
      FirstName=>"><script>alert('XSS')</script>&
      LastName=Peyton
      Email=lpeyton@site.uOttawa.ca
      Genders=0
      Approved=on
      Save=Save

Step 6
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 7

```
Scenario 3:  Update the current user profile
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/MemberProfile.aspx

Step 4
```

```
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton

Step 5: Attack 7
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
Input Form Fields:
      FirstName=%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e
      LastName=Peyton
      Email=lpeyton@site.uOttawa.ca
      Genders=0
      Approved=on
      Save=Save

Step 6
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 8

```
Scenario 3:  Update the current user profile
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/MemberProfile.aspx

Step 4
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton

Step 5: Attack 8
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
Input Form Fields:
      FirstName=<IMG SRC="javascript:alert('XSS');">
      LastName=Peyton
      Email=lpeyton@site.uOttawa.ca
      Genders=0
      Approved=on
      Save=Save

Step 6
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 9

```
Scenario 3:  Update the current user profile
Step 1
```

```
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/MemberProfile.aspx

Step 4
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
```

Step 5: **Attack 9**

```
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
Input Form Fields:
      FirstName=<IMG SRC=javascript:alert('XSS')>
      LastName=Peyton
      Email=lpeyton@site.uOttawa.ca
      Genders=0
      Approved=on
      Save=Save

Step 6
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx
```

Test Case: 10

```
Scenario 3:  Update the current user profile
Step 1
GET URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx

Step 2
POST URL: http://toh7.site.uottawa.ca/palis-test/Secure/Login.aspx
Input Form Fields:
      username=lpeyton
      password=lpeyton
      LoginButton=Log In

Step 3
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/MemberProfile.aspx

Step 4
GET URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
```

Step 5: **Attack 10**

```
POST URL: http://toh7.site.uottawa.ca:80/palis-
test/Teams/EditProfile.aspx?UserName=lpeyton
Input Form Fields:
      FirstName=<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114;&#1
05;&#112;&<WBR>#116;&#58;&#97;&#108;&#101;&<WBR>#114;&#116;&#40;&#39;&#
88;&#83<WBR>;&#83;&#39;&#41>
      LastName=Peyton
      Email=lpeyton@site.uOttawa.ca
      Genders=0
      Approved=on
      Save=Save

Step 6
POST URL: http://toh7.site.uottawa.ca:80/palis-test/default.aspx

-------------------------------------------------
```

Appendix C3 PAL-IS Test Execution Document

Document: Phase III Penetration Test Execution
Project: PAL-IS
Author: Pulei Xiong
Date: September 19, 2010

1. Purpose:

Execute the test cases created at the design phase (Phase II), and then analyze test results and create a test report.

2. Test Result Summary (First Round)

2.1 Application under Test

PAL-IS, version Beta 1

2.2 Test Report

```
------------- Test Campaign Summary --------------
# of test cases executed: 10
# of failed tests: 10
# of identified vulnerabilities: 2
# of entry points tested: 8
# of vulnerable entry points: 3
--------------------------------------------------

------- Web Security Repository Statistics -------
# of vulnerabilities: 5
# of attack vectors: 9
# of fuzz vectors: 26
--------------------------------------------------

----------------Test Result Details---------------
Test Case 1: launch an SQL Injection attack against <Log On> page
Result: Failed

Test Case 2: launch an SQL Injection attack against <Log On> page
Result: Failed

Test Case 3: launch an SQL Injection attack against <Search Patient>
page
Result: Failed

Test Case 4: launch an SQL Injection attack against <Search Patient>
page
Result: Failed

Test Case 5: launch an Stored XSS attack against <Update Profile> page
Result: Failed

Test Case 6: launch an Stored XSS attack against <Update Profile> page
Result: Failed
```

```
Test Case 7: launch an Stored XSS attack against <Update Profile> page
Result: **Failed**

Test Case 8: launch an Stored XSS attack against <Update Profile> page
Result: **Failed**

Test Case 9: launch an Stored XSS attack against <Update Profile> page
Result: **Failed**

Test Case 10: launch an Stored XSS attack against <Update Profile> page
Result: **Failed**

--------------------------------------------------

------- Identified Vulnerability Details ---------

Vulnerability: SQL Injection
probably exists on:
  Entry Point: ep002(Log On)
  Entry Point: ep005(Search Patient)

Vulnerability: Stored XSS
probably exists on:
  Entry Point: ep008(Update Profile)
--------------------------------------------------
```

3. Test Result Summary (Regression Testing)

3.1 Application under Test

PAL-IS, version Beta 2

3.2 Test Report

```
------------- Test Campaign Summary --------------
# of test cases executed: 10
# of failed tests: 5
# of identified vulnerabilities: 2
# of entry points tested: 8
# of vulnerable entry points: 3
--------------------------------------------------

------- Web Security Repository Statistics -------
# of vulnerabilities: 5
# of attack vectors: 9
# of fuzz vectors: 26
--------------------------------------------------

----------------Test Result Details---------------
Test Case 1: launch an SQL Injection attack against <Log On> page
Result: **Failed**

Test Case 2: launch an SQL Injection attack against <Log On> page
Result: **Failed**

Test Case 3: launch an SQL Injection attack against <Search Patient>
page
```

```
Result: Failed

Test Case 4: launch an SQL Injection attack against <Search Patient>
page
Result: Failed

Test Case 5: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 6: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 7: launch an Stored XSS attack against <Update Profile> page
Result: Failed

Test Case 8: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 9: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 10: launch an Stored XSS attack against <Update Profile> page
Result: Pass

--------------------------------------------------

------- Identified Vulnerability Details ---------

Vulnerability: SQL Injection
probably exists on:
  Entry Point: ep002(Log On)
  Entry Point: ep005(Search Patient)

Vulnerability: Stored XSS
probably exists on:
  Entry Point: ep008(Update Profile)
--------------------------------------------------
```

4. Test Result Summary (Regression)

4.1 Application under Test

PAL-IS, version Beta 3

4.2 Test Report

```
------------- Test Campaign Summary --------------
# of test cases executed: 10
# of failed tests: 1
# of identified vulnerabilities: 1
# of entry points tested: 8
# of vulnerable entry points: 1
--------------------------------------------------

------- Web Security Repository Statistics -------
# of vulnerabilities: 5
# of attack vectors: 9
# of fuzz vectors: 26
--------------------------------------------------

----------------Test Result Details---------------
Test Case 1: launch an SQL Injection attack against <Log On> page
Result: Pass

Test Case 2: launch an SQL Injection attack against <Log On> page
Result: Pass

Test Case 3: launch an SQL Injection attack against <Search Patient>
page
Result: Pass

Test Case 4: launch an SQL Injection attack against <Search Patient>
page
Result: Pass

Test Case 5: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 6: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 7: launch an Stored XSS attack against <Update Profile> page
Result: Failed

Test Case 8: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 9: launch an Stored XSS attack against <Update Profile> page
Result: Pass

Test Case 10: launch an Stored XSS attack against <Update Profile> page
Result: Pass

--------------------------------------------------

------- Identified Vulnerability Details ---------
Vulnerability: Stored XSS
probably exists on:
  Entry Point: ep008(Update Profile)
--------------------------------------------------
```