AUTOMATING BYPASS TESTING FOR WEB APPLICATIONS

by

Vasileios Papadimitriou
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science
Software Engineering

Committee:

______________________	Jeff Offutt, Thesis Director
______________________	Paul Ammann
______________________	Ye Wu
______________________	Hassan Gomaa, Chairman, Department of Software Engineering
______________________	Lloyd J. Griffiths, Dean, Volgenau School of Information Technology & Engineering
Date: ______________________	Summer Semester 2006 George Mason University Fairfax, VA

Automating Bypass Testing for Web Applications

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Vasileios Papadimitriou
Bachelor of Science
George Mason University, 2004

Director: Jeff Offutt, Professor
Department of Software Engineering

Summer Semester 2006
George Mason University
Fairfax, VA

Copyright © 2006 by Vasileios Papadimitriou
All Rights Reserved

Dedication

I dedicate this thesis to my wife Mimi and my son Konstantinos for their patience and support.

Acknowledgments

First, I would like to express my appreciation for the valuable advice and support from Dr. Offutt, who has been an outstanding advisor and professor throughout the course of this thesis and the MSSWE program.

For serving in my committee, I would like to thank Dr. Paul Ammann and Dr. Ye Wu.

Next, I would like to thank Human Resources Research Organization and specially Dr. Beverly Dugan, Dr. Carolyn Harris, and Dr. Cathy Stawarski for providing me with an excellent employment environment that promotes research and professional development, which had a great impact in the completion of this project.

I thank Wuzhi Xu, who's input in the early stages played a significant role in the foundation of this project.

Finally, I would like to acknowledge the valuable feedback that was provided by Christodoulos Christodoulou, Emmanuel Spyridakis, and Lima Beauvais.

Table of Contents

Page

Abstract 0

1 Introduction 1

2 Types of Client Input Validation 4

2.1 HTML Validation 4

2.1.1 Length Constraints 5

2.1.2 Value Constraints 5

2.1.3 Transfer Mode Constraints 6

2.1.4 Field Element Constraints 6

2.1.5 Target URL Constraints 7

2.2 Scripting Validation 7

2.2.1 Data Type Constraints 8

2.2.2 Data Format Constraints 9

2.2.3 Data Value Constraints 9

2.2.4 Inter-Value Constraints 10

2.2.5 Invalid Characters Constraints 10

3 Rules for Defining Test Cases 11

3.1 HTML Rules for Violating Input Constraints 12

3.1.1 Length Constraints Violation 13

3.1.2 Value Constraints Violation 13

3.1.3 Transfer Mode Violation 16

3.1.4 Field Element Constraints Violation 16

3.1.5 Target URL Violation 19

3.2 Scripting Rules for Violating Input Constraints 20

3.2.1 An Approach to Automate Scripting Rules 20

3.2.2 Current Support for Scripting Rules 21

4 Automating Bypass 23

4.1 Interface Parsing 23

4.2 Interface Analysis 25
4.3 Test Inputs 27
4.4 Test Case Generation 29
5 Empirical Evaluation 36
5.1 Experiment Design 36
5.1.1 Hypothesis 37
5.1.2 Independent Variables 37
5.1.3 Dependent Variables 38
5.2 Subjects 40
5.3 Results 41
5.3.1 ATutor Learning Content Management System 41
5.3.2 Joomla Content Management System 44
5.3.3 PhpMyAdmin, Web Based MySQL Client 47
5.3.4 Brainbench.com Online Assessment Products 49
5.3.5 Myspace.com Online Community Portal 51
5.3.6 NYtimes.com Online New 53
5.3.7 Mutex.gmu.edu Libraries Database Gateway 54
5.3.8 Yahoo.com, Global Internet Service 56
5.3.9 Barnesandnoble.com, Online Store 59
5.3.10 Amazon.com, Online Store 62
5.3.11 Bankofamerica.com, Online Banking 64
5.3.12 Comcast.net, Communications Provider 67
5.3.13 Ecost.com, Online Store 68
5.3.14 Google.com, Search Engine 69
5.3.15 Pageflakes.com, Community Personalized Portal 71
5.3.16 Wellsfargolife.com, Life Insurance 71
5.3.17 Result Summary 72
5.4 Confounding Variables 79
5.4.1 Effects of AutoBypass Implementation 79
5.4.2 Sample Web Applications Selected 81
5.4.3 Test Values Input 82
5.4.4 Result Evaluation 83
6 Conclusions 85

6.1 Future Work 86
References 88

List of Tables

Table Page

5.1 Experiment Subjects . . . 42
5.2 Result Legend . . . 43
5.3 Results for ATutor . . . 44
5.4 Results for Joomla CMS, Poll Administration module . . . 45
5.5 Results for Joomla CMS, Online User Information module . . . 47
5.6 Results for PhpMyAdmin, Current Database Control . . . 48
5.7 Results for PhpMyAdmin, Set Theme Module . . . 48
5.8 Results for PhpMyAdmin, SQL Form . . . 49
5.9 Results for PhpMyAdmin, Data Base Statistics . . . 49
5.10 Results for Brainbench.com, Information Request Form . . . 50
5.11 Results for Brainbench.com, New User Registration . . . 52
5.12 Results for Myspace.com, Event Search . . . 52
5.13 Results for Myspace.com, Music Search . . . 53
5.14 Results for NYtimes.com, Market Watch . . . 55
5.15 Results for mutex.gmu.edu, University Libraries Database Gateway . 55
5.16 Results for Yahoo.com, mail: notepad . . . 56
5.17 Results for Yahoo.com, compose message . . . 57
5.18 Results for Yahoo.com, Desktop Search Reminder . . . 58
5.19 Results for Yahoo.com, Weather and Traffic Search . . . 58
5.20 Results for Barnesandnoble.com, Shopping Cart . . . 59
5.21 Results for Barnesandnoble.com, Book Search . . . 61
5.22 Results for Amazon.com, Item Dispatcher . . . 62
5.23 Results for Amazon.com, Handle Buy . . . 63
5.24 Bankofamerica.com, ATM & Branch Locator by infonow.net . . . 65
5.25 Bankofamerica.com, Web Site Search . . . 67
5.26 Comcast.net, Service Availability . . . 68

5.27 Ecost.com, Shopping Cart 69
5.28 Ecost.com, Shopping Cart 69
5.29 Google.com, Froogle - Shopping Search Engine 70
5.30 Google.com, Language Tools 70
5.31 Pageflakes.com, User Registration 71
5.32 Wellsfargolife.com, Insurance Quote 72
5.33 Result Summary 73
5.34 Types of Invalid Responses 74
5.35 Result Summary by Violation Rules 78
5.36 Level of Effort 80

List of Figures

Figure Page

4.1 AutoBypass Architecture . . . 24
4.2 AutoBypass - Main Page . . . 25
4.3 AutoBypass - Form and URL selection for testing . . . 26
4.4 AutoBypass - Test Input . . . 30
4.5 AutoBypass - Test Results . . . 32
4.6 AutoBypass - Example Test Response . . . 34
4.7 AutoBypass - Example Mutant Form . . . 35
5.1 Exposure at Joomla CMS, Online User Information Module . . . 46
5.2 Fault/Failure at `Barnesandnoble.com`, Book Search component . . . 62
5.3 Exposure at `Bankofamerica.com`, ATM & Branch Locator . . . 66
5.4 Result Summary Graphs . . . 75
5.5 Responses by Violation Rule . . . 78

Abstract

AUTOMATING BYPASS TESTING FOR WEB APPLICATIONS

Vasileios Papadimitriou

George Mason University, 2006

Thesis Director: Jeff Offutt

By introducing new quality standards, the World Wide Web has a great impact on how software is being developed and deployed. Web software is mainly accessed through browsers and dynamically created user interfaces. HTML source and scripts are available to the user and can be modified and resubmitted due to the stateless nature of HTTP; thus, arbitrary requests from clients are permitted and web applications become vulnerable to input manipulation. Previous work on bypass testing is extended to develop an automated approach. An open source testing tool, HttpUnit, is used to build a prototype application, AutoBypass, which parses HMTL pages, identifies forms and their fields, and automatically creates bypass test cases that violate the user interface's constraints. AutoBypass performs testing on the external system level, eliminating the need for accessing the application source or server. The bypass method's effectiveness is empirically evaluated with web applications developed by professionals. The results show that applications generated numerous faults when bypass test cases were submitted. It is concluded that bypass testing can improve the quality of web applications by revealing potential vulnerabilities while providing an efficient method to reduce the development cost.

Chapter 1: Introduction

The World Wide Web has a great impact on how software is being developed and deployed. Web applications introduced new priorities for developers, driving the industry to value reliability, usability, and security instead of "time to market" which is more typical in the case of traditional software [13]. This shift on the criteria of software quality poses a new need in developing new methods to design, implement, and test software that is characterized by a distributed environment, implemented on a diverse collection of hardware and software platforms, and often requires interaction of heterogeneous components. Web applications are extremely loosely coupled and heavily user interactive in a dynamic manner, which is very different than the pre-established flow of control in traditional systems. User interfaces are dynamically created and the flow of control depends on the inputs provided to the system.

Interestingly, Xu et al. found in the 2003-2004 Common Vulnerability and Exposure (CVE) report that approximately 40% of the security problems originate from implementation errors that allow attackers to inject malicious code into carefully crafted inputs [19]. Due to the unique methods used to deliver graphical user interfaces and accept requests, web applications are extremely vulnerable to input manipulation attacks that compromise security. In addition, web applications are exposed to a variety of potential input sets that can reduce the reliability of the application.

Web software is mainly accessed through web browsers using HTML that allow the user to input data and submit requests. Input validation can be performed both

on the client and the server. In the past, client side validation was often used in order to reduce overhead network traffic between the client and server. In recent years, we see a trend to check input on the server as software components have more resources to perform input validation [14]. As a result, we find web applications using a hybrid of the two techniques to validate user input. In addition, client side technologies provide a rich environment for response to user events, improving the usability of the interfaces. For that reason, it is almost certain that client side validation will be used extensively, despite the need for more robust server side modules.

Client side input validation is performed by imposing constraints defined in the HTML language and the scripts supported by the browsers. On the other hand, server side modules are able to use a variety of programming languages and resources to identify invalid input requests.

Many features are available to the interface designers to control the way that users interact with the interface. By combining several HTML language features and scripting, interfaces can react to user's events and prohibit the form submission, if some requirements are not met, or provide feedback for the user. However, HTML code and scripts are available to the user once they have been accessed and can be modified and resubmitted to the server [14]. Due to the stateless nature of the Hyper Text Transfer Protocol (HTTP), server software cannot completely control user interactions; thus arbitrary requests from clients are permitted. HTML interfaces can be modified and/or circumvented, allowing the submission of unconstrained input to server software.

Several research papers have addressed this issue and identified ways that users can bypass the interfaces and provide input directly to the server. In fact, an active

area of research is SQL injection techniques, which essentially take advantage of the ability to bypass user interfaces.

This thesis, extends Offutt et al.'s work on bypass testing of web application, which defined a method for creating test cases for web applications that evade constraints set by the user interface [14]. The theoretical background has been revised and extended in order to support the use of an automated approach in the test case generation. A unit testing tool, HttpUnit [7], is extended to build a prototype software application that parses HMTL forms, identifies fields, create test cases that violate constraints set by the interface designer, and finally performs testing. This document includes analysis of types of client side input validation in chapter 2, definitions of the rules and pseudo-algorithms that are used to automatically generate test cases in chapter 3, the design of AutoBypass application in chapter 4, the experiment design and results and in chapter 5, and finally the conclusions in chapter 6.

Chapter 2: Types of Client Input Validation

Client side input validation is primarily performed by using HTML form controls, their attributes, and scripts that can access the Document Object Model (DOM) and evaluate the inputs or check for required fields before a form submission. A previous attempt [14] to categorize validation constraints categorized the two types of client side validation as syntactic and semantic. In this study the author revised and reorganized the individual items, which are labeled as HTML validation and scripting validation. Several aspects of the previous categorization are revised in order to provide a more abstract notion of the validation types and a better foundation for an automation tool.

2.1 HTML Validation

Types of input validation that are performed using the HTML syntax on forms are listed in this section. HTML has syntactic rules that define input controls that implicitly validate the user input. Although web browsers define different rendering capabilities for HMTL pages and forms, the HTML 4.1 language specification [16] is used to identify specific features that are applicable to this research. The following types of input validation are defined:

1. Length Constraints

2. Value Constraints

3. Transfer Mode Constraints
4. Field Element Constraints
5. Target URL Constraints

2.1.1 Length Constraints

The most common type of input constraint imposed by the HTML syntax is the *maxlength* attribute for text or password input controls. This attribute limits the number of characters of the string entered by the user in the input field.

2.1.2 Value Constraints

The HTML language specifies input controls that apply limitations on the set of values available to the user. Parameter name and values pairs contained in a form can be submitted either explicitly or implicitly. For instance, the number of selections of a drop down list or check boxes is limited by the predefined set of values applied by the application designer; these controls explicitly constrain the set of values available to the user. On the other hand, hidden fields include predefined values without the user's knowledge and implicitly dictate the values available to the form. The types of controls that define value constrains are:

- Hidden controls (`input` element)
- Menu controls (`select`, `option group`, and `option` elements)
- Boolean controls (`input` elements of type `checkbox` or `radio button`)

- Read-Only controls (`textarea` elements or text controls with the `readonly` attribute set)

Radio buttons allow mutually exclusive selection, while checkboxes can allow multiple selections. Similarly, the select element allows single or multiple selections.

2.1.3 Transfer Mode Constraints

The form element specification includes the method attribute that defines the Hyper Text Transfer Protocol (HTTP) method used to submit the form. There are several request methods defined on HTTP, but this study only uses the two most common, GET and POST. Note that in the HTML 4.1 specification, the method attribute is not required but forms that do not define a submit method will be submitted using the GET by default. Forms that use the GET method are transmitted in ASCII as parameters appended to the URL as a form data set separated by a question mark (`?`). GET request parameters and their values are clearly visible to the user in the URL, and they impose a maximum limit on the data transmitted of 1024 bytes. POST requests are constructed as HTTP messages to the action URL, can have specified data encoding and are not limited in size. Users can bypass the HTTP method that is imposed by the interface simply by changing the HTML attributes of the form.

2.1.4 Field Element Constraints

Interfaces in web applications provide a defined set of controls including inputs, buttons, and menus. The set of controls submitted to the server application is predefined by the application designer and classifies the set of controls in a form as input constraints. Values are explicitly assigned in controls that render on HTML pages and

implicitly in non-rendered fields such as hidden controls. The HTML specification does not require all controls in a form to be successful (transmitted to the server). Instead, during form submission, all the successful controls are organized in a package and sent by the user agent (typically a web browser). On the other hand, web applications can implicitly constrain the user from supplying inputs in specific controls on the forms. This type of input constraint is also related to the scripting validation types.

2.1.5 Target URL Constraints

The most common mechanism to traverse through multiple web pages are links (anchor `<a>` tag). All links in HTML pages generate GET requests to the web servers; therefore, anchors allow URL rewriting by appending parameter-value pairs to the URL. Similarly to the GET requests, links with parameters can be modified by the user by simply altering the HTML code or modifying the address of the target URL. Through a predefined set of links, applications can constrain the user to navigate to a fixed set of URLs [14].

2.2 Scripting Validation

HTML pages may contain scripts to respond to user events and validate data input prior to form submission [14]. There are multiple scripting languages available, such as JavaScript, TCL Script, and VBScript, to perform similar actions, but the use of JavaScript is dominant and was used in this study. The main advantage of using scripts in HTML pages is the full access to the Browser Object Model (BOM) and the Document Object Model (DOM). Specifically, access to the DOM allows programmers

to control and respond to following events: *onload, onunload, onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onfocus, onblur, onkeypress, onkeydown, onkeyup, onsubmit, onreset, onselect,* and *onchange* [16].

By associating actions with form elements, application designers can respond to user inputs, prohibit form submission due to input restrictions, add/remove elements from the forms, change the form's action etc. Similarly to the HTML validations, a set of validation types are defined, which are enforced via scripting. Scripts can perform all the types of validations that are implicitly done by the HTML syntax, yet scripting provides ways to enforce application specific requirements and apply semantic restrictions. Via scripting, a web interface can perform all the validation checks described for HTML constraints; however, validation with semantic scope that is available with use of a scripting language cannot be replaced by the interface itself. Several types of scripting validations:

1. Data Type Constraints
2. Data Format Constraints
3. Data Value Constraints
4. Inter-Value Constraints
5. Invalid Characters Constraints

2.2.1 Data Type Constraints

A major use of scripting validation is to examine input values that relate to specific data types in the server application. By definition, all values in form controls are

transmitted as sequence of characters; therefore, validation is required to ensure that the server side software component receives appropriate values. For instance, a field that accepts inputs for age should only accept integers. A scripting function can verify that requirement by checking the input control before submission and ask the user to correct mistakes.

2.2.2 Data Format Constraints

Often, applications require input for values that need to conform to a specific format. For instance, in the US, a zip code is a string of five digits and a phone number is typically comprised of the area code and the local number (10 digits). Other types of values that are often checked are the format of money, personal identification numbers, email address, URLs etc. Using client side scripts to validate these types of inputs in the interface applies constraints that can be circumvented by a user; all the user needs to do is save the HTML, modify to delete scripts and resubmit the invalid data.

2.2.3 Data Value Constraints

Data value constraints are used in web applications to provide input validation in fields that have semantic restrictions. As an example, a field that denotes the age of an individual should not only prohibit non-integer values (as a data type constraint), but also limit the value of the integer to an appropriate range (e.g. 0 - 150).

2.2.4 Inter-Value Constraints

A very useful function of scripting in web interfaces is to constrain relationships among several input controls in a form [14]. For example, when payment information is required on a form, a choice of a credit card payment should require the input of credit card type, account number, and expiration date. Omission of any of these values makes the transaction impossible to complete; therefore, scripts are used to verify that all required fields are completed and valid prior to form submission.

2.2.5 Invalid Characters Constraints

Security measures require input validation that will prohibit malicious string injections in to web applications. Inputs are frequently checked on the client to determine whether they contain obvious invalid strings that can cause security infiltrations on the server. Inputs with invalid characters can not only affect the security of the application, including SQL statements, but also the robustness of the web applications. Several types of invalid characters, such as XML tags (`<, >`) and directory separators (`../`) can cause server applications to become unavailable. Wheeler provides a list of common invalid characters that affect software [17].

Chapter 3: Rules for Defining Test Cases

Having identified the types of the client side constraints that are imposed by web interfaces but can be bypassed by the end user, this chapter defines a set of rules that are used to generate invalid inputs. The generated test cases are used to test web applications with inputs that are validated on the client side either explicitly by the application or implicitly by the interface. Following the structure of the validation types defined in the previous chapter, the author aims to structure a set of rules and algorithms implemented on the automated test generation tool.

A challenging issue for automatic test case generation is how to provide new test values for fields and parameters that violate the applicable constraints. Theoretically, there are an infinite number of possible new values and parameters that can be submitted. Yet, for appropriate testing, values that exist within the domain of the application are preferred. Assume a value constraint applied through a drop down list that allows the user to select his/her favorite color; the existing list could be blue, yellow, and green. An effective test would be to add the value of red since it is part of the input domain and it is more likely to affect the application. However, the problem is knowing alternative values that are in the domain.

The author refers to this condition as the “Semantic Domain Problem” (SDP). The SDP can be approached in many different ways. Random values can be used to test the application; however, this approach is inefficient as it can lead to an infinite number of alternative values, which are unlikely to be part of the application domain.

Therefore, this approach is not used in this research. Alternatively, automatically generated *domain related* values can be used. By examining the initial values of the input set, one can potentially generate related values that belong in the same domain. For example, in a drop down list that allows course selection (e.g. SWE763), a pattern can be observed and be used to generate new values. In this case, the pattern is 3 characters followed by 3 digits. An interesting approach is one similar to Google sets (`http://labs.google.com/sets`), where with an input of a few terms, one can generate a set (small or large) of values with in the same domain. For instance, the input of values red, blue, green produces: Blue, Red, Green, Yellow, Black, White, Magenta, Cyan, Gray, Browser, Orange, Purple, Brown, and Transparent. This approach is very promising, yet the author leaves its implementation for the future. Another solution is to gather parameter and values from one application by parsing multiple pages that would provide input with parameters used. Finally, tester supplied values can be used to test web applications. This approach is feasible for this study, and also appropriate for selecting new values that can be intelligently chosen by human users that are familiar with the application domain. Nevertheless, this reduces the automation of this process.

3.1 HTML Rules for Violating Input Constraints

Parsing HTML code allows detection of constraints that are mainly enforced by the web interface. The following subsections define a set of rules that are used to generate test cases with regards to the HTML client validation.

3.1.1 Length Constraints Violation

This type of validation is the most trivial to detect and bypass. For each text or password input in a form with an attribute *maxlength* = L, the test case generated (TC) is length = L+1. On the other hand, automatically selecting an effective value that will violate the length constraint is not trivial. As described in the beginning of chapter 3, the SDP applies, and a human tester is required to provide test inputs.

In the automated test generation tool, the form controls of the subject are scanned to find maxlength attributes. Disabled controls are excluded in this process since they are not successful controls upon a form submission. Then, the input domain is searched for values assigned to the same parameter that would violate the input length. If none are found, the search continues to values that are assigned to other parameters in the application input domain and violate the length constraints. If still no value satisfies the test case requirements, a random string is generated and replaces the default value of the control.

3.1.2 Value Constraints Violation

There are several ways that test cases can be generated by the value constraints on an HTML form. In general, this type of constraint is violated by submitting a value outside the predefined set of values (modified value) [14]. Modified values are generated by either adding new values (as described in the beginning of chapter 3) or by value omission, which is applied to obtain values outside the predefined set. For all input, selection, and button elements specified in the HMTL language, the value attribute is of type `CDATA` [16], which is a sequence of characters (or string). The preset values can be violated by replacing the existing values with an empty

string. Value omission applies to hidden, read-only, selection menus, radio buttons, and checkboxes controls and can be accomplished for each control as follows:

- For each Hidden Control in a form, the original value is replaced with a modified value. Use of hidden controls is very sensitive, as many applications use hidden fields to identify the user or otherwise transmit values between server side software components [14].
- For each Read-Only Control, the *readonly* attribute is omitted and its value is set to a modified value.

For selection menus, radio buttons, and checkboxes, we define an abstract way to produce test cases. By specification, form submission includes name-value pairs and disregards the rendering scope of the controls themselves; thus, regardless of the type of input, only the parameter name and value(s) will be transmitted to the server application. An efficient testing method for the preset values in this set of controls requires an abstract model that will distinguish them into single-value and multi-value controls. Single-value controls include selection menus that prohibit multiple selections, radio button groups and checkboxes that do not belong in a group. Multi-value controls include checkboxes that are grouped by using the same identifier, and selection menus that allow multiple selections. Check boxes that share the same name identifier will allow multiple selections, and thus behave similarly to the multi-value selection menus. With this abstract concept in mind, two more rules for test generation are identified:

- For each Single-Value Control, the original value is replaced with a modified value.

- For each Multi-Value Control, a combination of a valid value and a modified one is tested. The control is submitted with pairs of valid and modified values, but producing all the combinations of the possible valid-invalid sets is avoided due to the potential for generating an enormous number of test cases, which is likely to produce similar results.

There are some special types of input types that are related to value violation rules. These are *disabled*, *image maps*, and *file upload* controls. Disabled fields also have preset values by the interface; yet disabled controls are not successful during form submission (i.e. not transmitted to the server). Disabled fields can be enabled on the client side by manually modifying the HTML code; yet, this is addressed in the *Field Element Constraints Violation* rules. Therefore, disabled fields are out of the scope of this rule.

Next, image maps are primarily used for navigation [16]. Moreover, prior to the availability of advanced technologies for animated graphical elements (such as Macromedia Flash), image maps were also used to provide visual effects rather than to send critical data to web applications. According to the HTML specification, images can be used in several ways with the same outcome. The input element can be of type *image* providing the functionality of a button. The use of image buttons will result to the submission of the form data appended with the control's name and its x and y pixel coordinates relatively to the image location. Alternatively, an anchor combined with map element can be used to specify areas of an image that will function as different references. Maps can be defined as client side or server side. Client side maps use predefined references specified in each area and become equivalent to an link. For server side maps the browser will generate a GET request to the server by

appending the coordinates of the mouse location as parameters.

Lastly, input controls of type *file upload* are used to upload files from the browser to the server. The specification defined for file upload at RFC1867 [11] requires the HTML form ENCTYPE set to multipart/form-data, which is used by most web technologies. It is unlikely that manipulations of this input control will produce major faults to the applications, because of the fact that file uploads are not very common, and second because file uploads require specific processing by the applications, which will force developers to eliminate invalid requests. However, in theory one can violate the constraint by providing corrupted files, files with different extension than those required, try to upload malicious code, or simply replace the value of the parameter with a string.

3.1.3 Transfer Mode Violation

Violation of the transfer mode constraint is accomplished by alternating the method of HTTP transfer. The forms that define a GET method will be replaced with POST and vice versa.

3.1.4 Field Element Constraints Violation

The rules for violating field element constraints are very similar to the rules for value constraints. With the bypass method, field element constraints are violated by submitting forms that include a set of controls that are different from the predefined selection. For example, an HTML form that consists of a user name, test input and a password input has a defined set of two inputs. To violate this constraint, one can remove or add a new field. Specifically, modified sets of controls are created first by

control omission, then by control addition, and finally by control type alteration. An exception to this rule and the implementation of the automation tool are the image maps and file upload controls for the same reasons discussed in section 3.1.2.

Control Omission

To achieve modified control sets by control omission, form controls are eliminated sequentially. Theoretically, for a form with n controls, a large number of test cases can be generated as the sum of the all possible combinations for a set of predefined controls, as shown in equation 3.1. For instance, for a form with 3 controls, first one at a time is omitted, then two at a time and finally an empty form is submitted. That is, 3+3+1 = 7 modified forms can be used. Instead, only one form control is eliminated sequentially in order to avoid a large number of test cases that are expected to have similar effect.

$$|TC| = \binom{n}{1} + \binom{n}{2} + \binom{n}{3} + ... + \binom{n}{n-1} + \binom{n}{n} \tag{3.1}$$

Control Addition

Next, test cases are generated by form control addition. For effective testing, the added fields must exist in the current application, and perhaps used in other forms. In the case of random control addition, the server-side software will most likely ignore them. Adding new controls is affected by the Semantic Domain Problem; therefore, controls are primarily added by a human tester that can identify possible new controls that belong to the application domain. Apart from the user input, the automated tool will implicitly use disabled controls and other buttons defined in the form that

are not used for submission. Such controls may or may not be available.

Control Type Alteration

Field value constraints are also violated by virtually converting the type of the field elements. Since all successful controls in a form are transmitted as name-value pairs to the server, alternating a control type will have no effect. For instance, the data set will not change if a radio button is altered to a checkbox (as long as the input name remains the same). Taken this fact under consideration, instead of changing the HTML input types of the controls (e.g. from radio to select), only controls that accept single values are modified to accept multiple values and vise versa.

It is assumed that omitting a selection from a multi-value control will not significantly affect the application, since most applications use such controls for optional additional selections by users. On the other hand, adding values to originally single-valued parameters is expected to influence the tested application since the application will expect to receive only one parameter. Therefore, the automated tool implements this rule by adding a value to every single-valued parameter, which alters its type to a multi-value parameter.

The main focus for testing the application will be values from the human tester; thus, the first candidate values are those in the input domain. If no input is provided, the default values of other controls in the form expecting that values that already exist in the application are more likely to affect the output. Yet, in a form with k controls we can create k-1 test cases for each and k(k-1) in total; i.e. for each control we would have to add one default value from every other control. That would result in a large number of test cases that are likely to have the same effect. For that reason

the automation tool limits the additional values to the inputs by the tester and an additional one found in a default value of an other control in the form.

3.1.5 Target URL Violation

Target URL inputs typically exist in forms using GET methods and requests embedded in links with parameters. By altering the parameters and their values, testers can create test cases that will interact with the state of the server as unexpected requests are received. For each anchor that contains name-value pairs, combinations of modified parameter sets can be created. Modified parameter sets can be generated either by dropping parameters (equivalent to omitting fields or values) or using alternative URL paths. Alternative URLs must exist in the web application, probably pointing to different components in the same application domain, in order to be effective. Addressing this requirement programmatically will introduce SDP since a large number of random inputs can be generated that will possibly have no effect. Instead, this version of the testing tool uses name-value pairs supplied by a human tester.

In addition, in a given HTML interface of an application there may be several URL links that reference and send parameters to different application modules or even different applications all together. By testing all possible URLs that carry parameters, a great number of test cases will be created. Just to illustrate that point, the `bestbuy.com` index page contains 187 URLs, and `amazon.com` contains 347. Moreover, the test set will address modules from the application that are not in the current focus of the user, or may even belong to a different application all together.

To address this issue, the automated tool was implemented to form groups of

URLs found according to the application (identified by the URL domain) and application component (identified by the URL path). The *java.net.URL* class is used to reconstruct and evaluate the application and component references, in addition to the validity of the URL format. During that process, links that include no parameters are isolated together with links that use scripts, those that include relative paths that are not correctly resolved (e.g. *../../somerelativepath.html*), and finally those that produce exceptions for unexpected situations. The user may select from the remaining groups that may include one or more URLs for each application and component to generate URL violations.

3.2 Scripting Rules for Violating Input Constraints

This implementation of the testing tool does not include an automated mechanism for the violation of constraints that are enforced via scripting in web application interfaces. Yet, on the theoretical component of this study, a possible approach for automating scripting violation is defined and presented below. In addition, current functionality of the testing tool that can be manually be used to address some of the scripting violations is also discussed.

3.2.1 An Approach to Automate Scripting Rules

To approach the detection of scripting validation, scripts running on web interfaces must be parsed. Indeed, an automated detection of the semantic validation via scripting is non-trivial. The initial strategy to generate test cases that aim to violate constraints for all types of scripting constraints types (Data Type, Data Format, Data Value, and Invalid Character). Upon parsing the scripts, scanning to detect error

states (e.g. `alert` statements in functions that are triggered upon form submission) can identify input restrictions. The tool may then detect the conditions under which the error states occur and create test cases that include values that violate the interface's validation.

Another strategy refers to inter-value constraints that occur when a form element is changed. A user action may trigger the change of other fields and their values in the form. For example, assume a web site about cars; a form may include a drop down list for users to select a make and a drop down list to select a model of a vehicle. The selection on the car make list may affect the values displayed on the model list dynamically. Often a JavaScript manipulates the form fields on the client side and the inter-value constrains are enforced. An approach for automating the violation of such cases would be the detection of scripts that alter values on fields upon events (e.g. `onchange`) and comparting values of other fields. A sophisticated mechanism is required to model the relationships of values in various fields and then create test cases that will violate them.

Finally, a very common example to enforce scripting validation is checking for the required fields in a form. To ensure that all required fields have inputs by the user, scripting functions check the form contents prior to form submission. The approach to violate them is to first identify all required fields dynamically, then create a control set that has no values, and finally submit values that violate other constraints.

3.2.2 Current Support for Scripting Rules

The automation tool is designed to behave similarly to a browser that does not support scripts. In such cases, applications are by default exposed to invalid inputs as scripts

that perform the validations are not enabled. Using application domain knowledge, a tester can manually select test values that violate the scripting constraints in the interface and essentially achieve the same results as an automated scripted constraint detection and input violation.

The tester is expected to be familiar with the interface and the constraints enforced. During the process of creating the test cases the tester has the ability to address these issues and invalidate the requests. For instance, assume that an input field exist to enter the user's age. Typically, a script function is used to validate and warn the user if the data entered is invalid type or out of range; e.g. '*-1*', or '*230*', or '*someText*'. In that case the human tester can simply test these values by providing additional parameters in the tool's interface and be part of the HTML field value constraints.

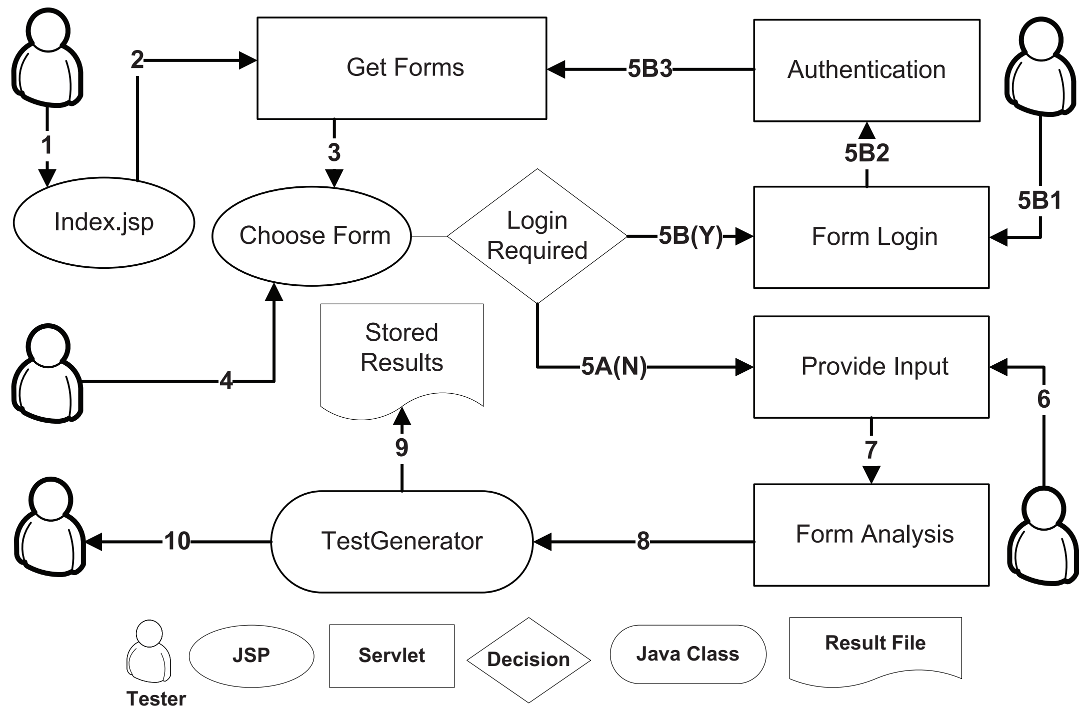
An additional feature was provided in the user interface of the tool that allows the tester to efficiently violate input with invalid characters. For each input control, the user can select a category of invalid characters, including empty string, commas, directory paths, ampersands, strings starting with forward slash, strings starting with a period, control characters, characters with high bit set, and XML tag characters. These sets of invalid characters are then used to generate test cases. Actually, special characters created problems in the testing tool. In earlier versions of the testing tool, when invalid characters were selected, the generated test cases were corrupted and problems occurred with the display and scripts used for the tool's interface. For that reason, the tester's selections of invalid characters had to be encoded with tokens and decoded again, just before the test case generation mechanism assigned them to the parameters of the test cases.

Chapter 4: Automating Bypass

Test case generation, as defined in the previous chapter, is performed by an automated tool, AutoBypass. The tool is an extension and modification of HttpUnit, an open source package. HttpUnit is written in Java and emulates web browser behavior by allowing Java programs to manipulate HTML elements, submit requests and receive responses that can be parsed and examined [7]. A new approach was followed to modify HttpUnit and adapt it to the functionality required to perform automated test case generation based on bypass methods. By itself, HttpUnit simulates the behavior of a web browser, yet it requires manual creation of test classes that will test specific forms, fields and URLs that are known in advance. In the case of Bypass testing, a dynamic approach to parse HTML pages needed to be designed, since that form information is not available until the interface is parsed and analysed. Figure 4.1 shows the interactions of the user and the various components of AutoBypass. The functionality of the tool is defined in the following sections.

4.1 Interface Parsing

As a first step, a tester provides a URL for the application to be tested. This initial screen is shown in figure 4.2. Upon submission, the tool parses the input and checks whether it is a valid URL. The document corresponding to that URL is received and analyzed. All anchors, forms, and script elements are obtained and made available to the application.



Sequence	Event
1	The tester provides a URL for the application to be tested
2	All anchors, forms, and script elements are obtained and made available to the application
3	The request is forwarded to a component that presents the forms and the URLs found
4	The tester is presented with a list of forms and URLs available to select for testing
5A	No authentication is required (proceed to seq. 6)
5B 5B1	The tester chooses to test password protected components The tester provides the authentication information for tested application
5B2	Submit login name and password to the application. A session is established by the WebConversation object and the subject application (Return to Seq. 4)
6	The tester provides test input
7	Bypass rules are applied to create test cases
8	Submit invalid requests to the application
9	Test responses are stored to file
10	Result summary presented to the tester

Figure 4.1: AutoBypass Architecture

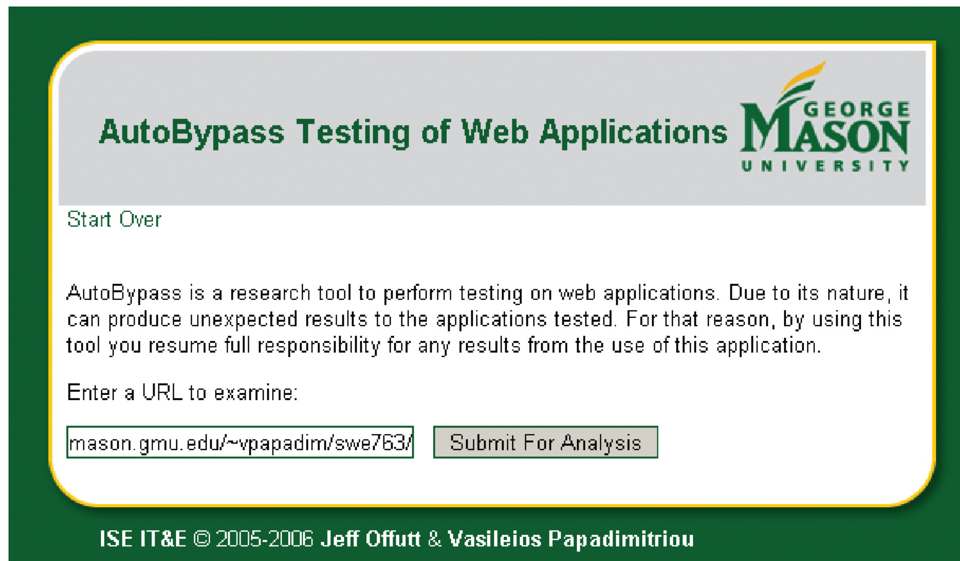


Figure 4.2: AutoBypass - Main Page

4.2 Interface Analysis

Given a valid URL for the subject application, the request is forwarded to a component that presents the forms and the URLs found. Figure 4.3 shows a sample output for this stage. The tester is presented with a list of forms available and required to select the form that test cases will be generated for this set. In addition, URL references (links) are identified and processed in order to perform optional test case generation as described in chapter 3.

At this point, the application also detects whether the forms that were found include login fields. That is decided by parsing the field names (e.g. login, user name etc.) and also by detecting password controls. If the tester selects to test password protected components, he or she may choose to provide authentication information. The authentication process takes place through submitting login name and password

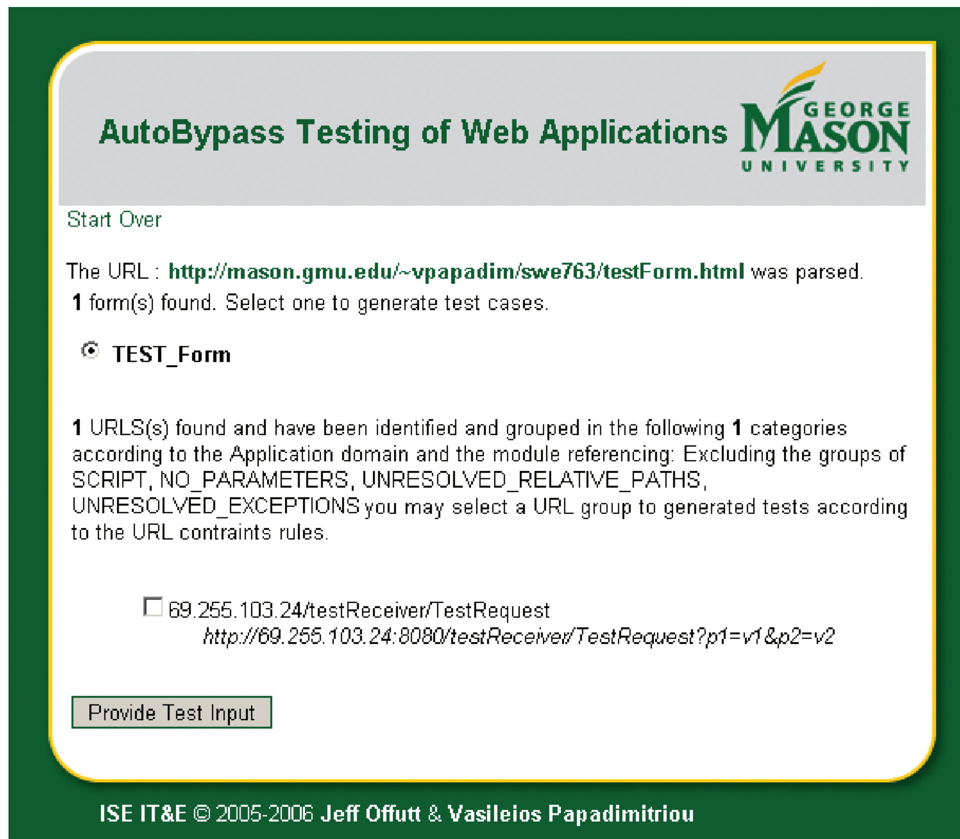


Figure 4.3: AutoBypass - Form and URL selection for testing

to the application. A session is established by the HttpUnit.WebConversation object and the subject application. Yet, protected areas present a different set of input forms to the users. For that reason, AutoBypass forwards the request back to step two, where the new output of the tested application is analyzed. The tester is then presented with a new set of forms and URL groups that can be analyzed. In addition, a new field is available to give the option for testing a different area of the web application that that was not accessible prior to authentication but now can be reached.

4.3 Test Inputs

The selected form is analyzed and all controls are identified. The interface shown in figure 4.4 is generated based on the subject's form fields, and allows the tester to provide test inputs. As discussed in chapter 3, the set of parameter-value(s) pairs that will be transmitted to the server for the generated test cases should be all valid with the exception of the modified parameter in each test case. For that reason, the tester shall provide a complete set of valid parameters (a set that would result in a known valid response conforming to the constraints of the HTML interface), and a set of invalid parameters that will be used in the invalidated requests.

For every control, the default values from the HTML specification are loaded automatically if available, and can be changed. The default values will constitute the valid set for this parameter. The user will then have the opportunity to specify additional values that violate the interface constraints. For instance, given an interface with an input control having a *maxlength* string constraint, the tester can specify a string larger than would normally be allowed. The tester can enter additional values

separated by commas, which will define the pool of new options for use in the bypass testing automation. Invalid inputs can also be specified by selecting invalid characters (empty string, control characters, etc).

Some variations for the tester input values occur for the disabled controls and buttons. Disabled controls would not be successful and thus their default values would not be transmitted to the server. For that reason, the default values of disabled items, if available, are automatically transferred to the test value set for this parameter. Buttons in HTML forms create another challenge for the automation of bypass testing. A form may have more than one button that can be used to submit the request and can carry different values that will affect the processing in the server. Thus, it is unrealistic to use all buttons and their values in the default parameter-value pairs. The tester chooses which main button will be used as the standard in the test set. If no button is specified, the application will arbitrarily pick one. The remaining buttons will be used as part of the input domain for other violation rules (e.g. control addition).

Finally, a field for additional parameter-value pairs is provided. Using this text box, the tester can specify additional input parameters and values. A specific format is required to allow multiple parameters that may have zero or more values (e.g. simulating a select control with multiple selection). The format required is a parameter name followed by a colon ':' and its value set. Value set may include zero or more values separated by the dollar sign ($). Parameter name-value pairs are separated by semi-colons. For example:

```
name1:value1;name2:value2a$value2b;name3:;
```

4.4 Test Case Generation

When the collection of the control elements and their test values are constructed, bypass rules are applied to create test cases, generate invalid requests and finally submit them to the application. A java class, *TestGenerator* (TG), was implemented to generate and run the test cases. Every new test set will generate a new instance of TG given a set of parameters that would specify the test case generation, such as the *WebResponse* and *WebConversation*, which are objects that contain the subject forms and links from its initial response and current HTTP session, a *Map* with the default values for each control, a *Map* with the input domain that includes the test values provided by the user, a *Map* with the buttons to be used for submission and other operational parameters.

The map representation of the parameter value pairs is initially a concept borrowed from the Java servlet API [2], which provides the function *getParameterMap* to retrieve the request's parameter mappings. This function returns a *Map* containing parameter names as keys corresponding to parameter values. In this API, the keys are represented by a *String* and the values by *String Array*. Note that for each parameter in a HTTP request, there may be more than one value (e.g. drop down fields that allow multiple selections) that are stored in the value array. In the AutoBypass implementation, a *Set* data structure was used to represent the parameter's values, instead of the string array to allow a flexible environment for value addition and omission, as well as to create a homogenous approach for all the test data collected (default values, invalid parameters, URL parameters, etc.).

Figure 4.4: AutoBypass - Test Input

As each of the bypass constraint violation rules is applied, a *Map* of parameter-value pairs is generated. A member function is implemented to accept the mappings that represent the invalid request. This function creates a simplified HTML form equivalent to the test case specification, stores it in a local HTML file and then requests a *WebResponse* object using the HttpUnit API.

The generated HTML represent an invalid *mutated* version of the valid parameter-value set. The *WebForm* is then retrieved in the response and added in the a global *Vector* that stores all the test cases waiting to be submitted. Upon completion of the test case generation, TG uses the *runTests* method, which iterates through the vector that contains the mutant forms and submits them to the subject application using the pre-established session. For each test case, the test response is stored in a file along with the test case information (including a public URL for the mutated form and response and the form control related to this test case with its original and modified values).

The tester is presented with the final screen of the application that includes the test result summary. An example is shown in figure 4.5. The tester can review a table of all the tests generated and use links to the subject's responses to review each test case's result. Together with the the result table, the tester can also access logs that will provide further information, including the test generation log, the input domain map, the default value map, the additional controls specified, the invalid characters selected for each control, and the buttons used.

A major issue with the automation of web application testing is still the review of the application responses. Indeed, the bypass methodology produces a variety of mutated requests that are not expected to result in the some predefined application

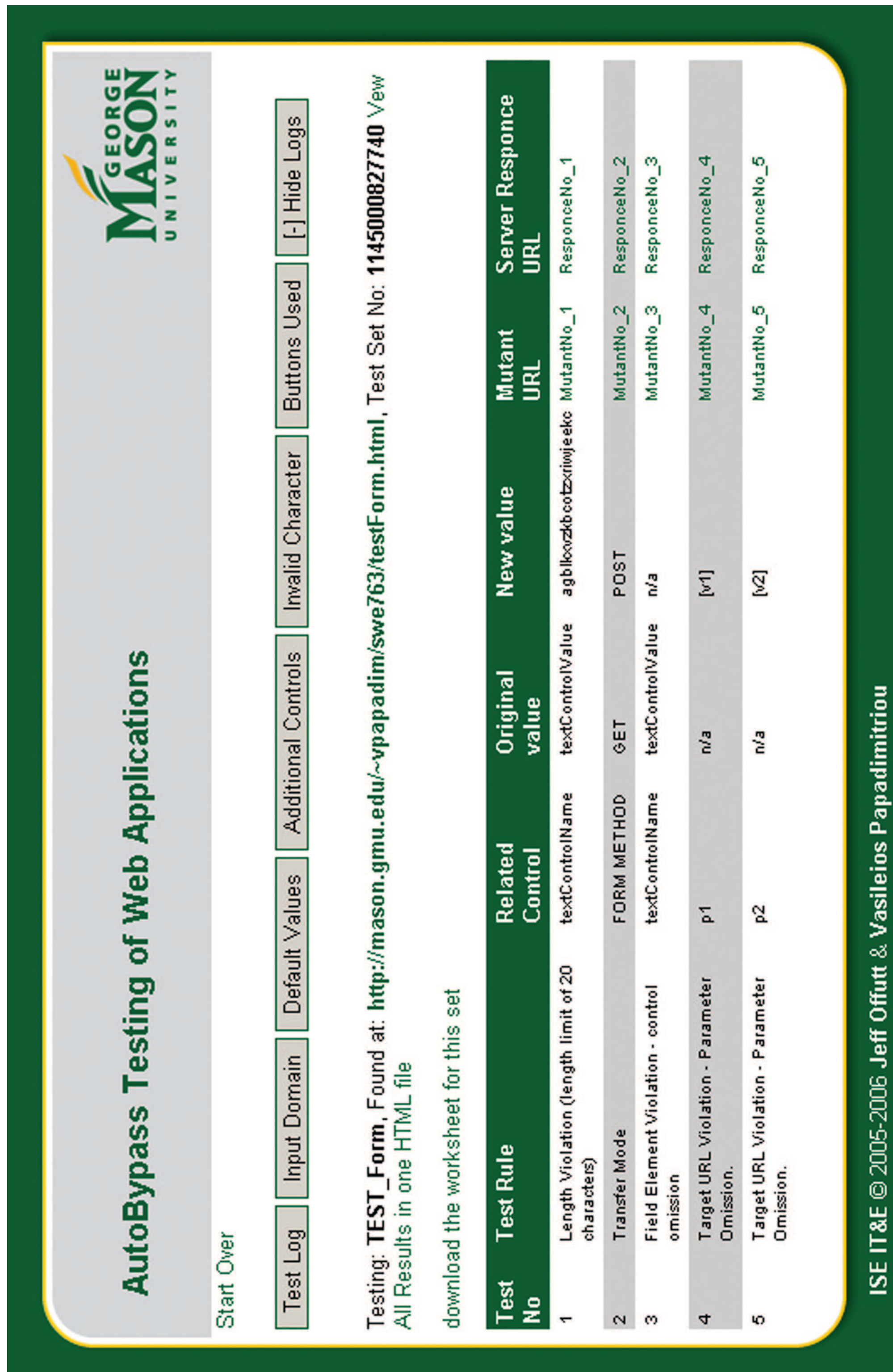


Figure 4.5: AutoBypass - Test Results

output. The usual approach used in unit testing frameworks is to compare the results with a predefined output set given a predefined input set. This is not the case with the automation of bypass, since that invalid requests may produce different responses. Therefore, the review of the results must be performed by a human that will have to account for the mutations created and whether the response of the application was appropriate. In this process, the two generated HTML documents for each test case (mutant form and the subject's response) are of special importance for the review and for that reason, they are both stored to a local disk with web access permissions to allow later access for evaluation by the tester.

The result files follow a specific structure with four parts as shown in the example on figure 4.6. The first part, on top, includes the URL of the subject application in which the form tested was found, in addition to a link to the mutant form that was used to produce this response. The second part includes the test case description with the rule information, the test case number, the related control, the original value(s), and the modified value(s). The rule used to generate this test case may include additional information. For instance, in a length constraint violation, the control's *maxlength* is reported.

Note that in special cases, original and modified values may not be applicable; such examples include the control omission rules that will not have a modified value since they are omitted all together, and the control addition which will not have an original value. The third part consists of the mutant form control list along with their values as submitted. The last part of the result output is a capture of the HTML response from the subject application. It is important to note that other supporting elements, such as that images and external script files, are not saved with the result.

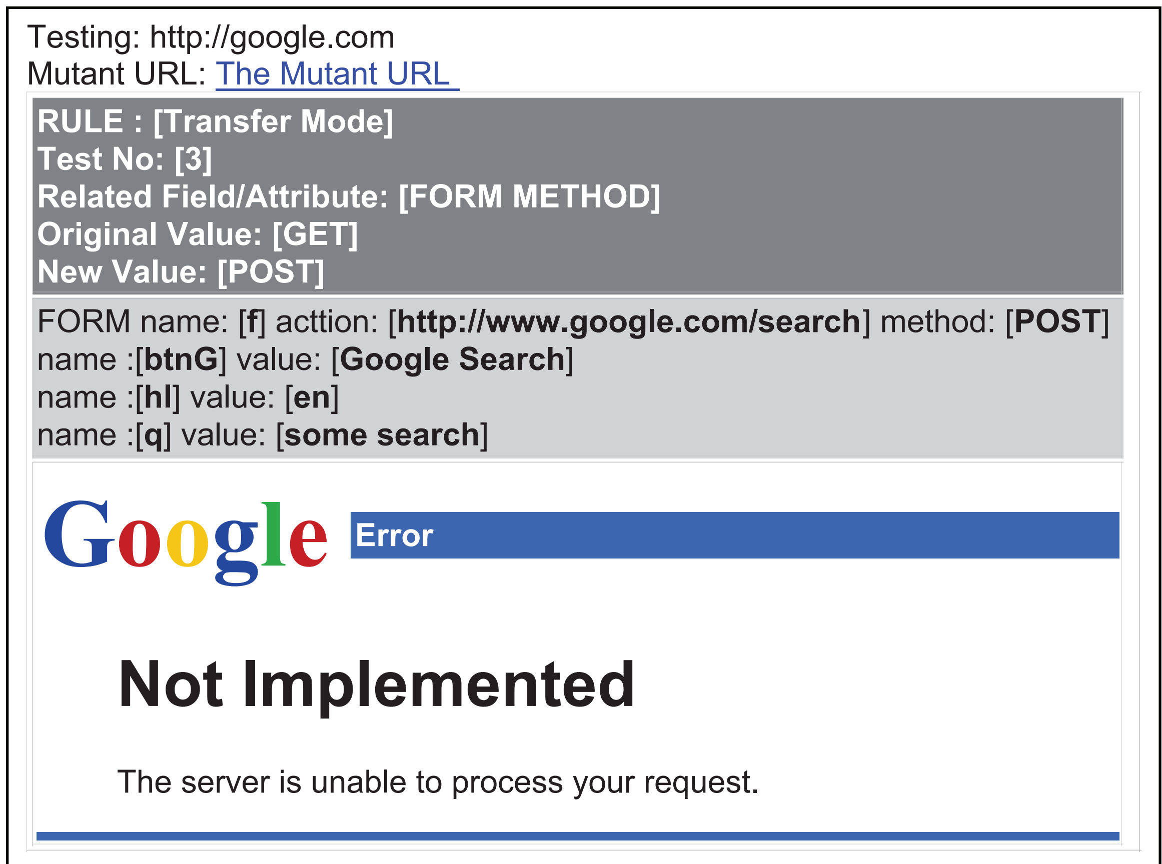


Figure 4.6: AutoBypass - Example Test Response

theForm.getBaseURL(): http://www.google.com/
theForm.getAction() :/search

btnG: Google Search
hl: en
q: some search

Submit This Mutant

Figure 4.7: AutoBypass - Example Mutant Form

That may produce some display limitations that should not affect on the evaluation of the results.

The mutant forms provide the ability to verify the result by accessing the subject with a common browser. During the development of the automated test tool, it was very important to have access to a copy of the mutant forms and verify the automated responses. Thus, an HTML version of the mutated form is included in the test case's result package. For instance, the mutant form for the response discussed above can be accessed with a browser and resubmitted manually. Figure 4.7 shows the HTML version of the mutant form.

Initially, a generic HTML submit button was introduced that had a name and a value that were transmitted along with the mutated request. For that reason, a scripted submission is provided to avoid the addition of parameters out of the test case scope.

Chapter 5: Empirical Evaluation

A goal of this project is to use the theory of bypass testing to implement an automated tool that can generate test cases and provide developers with an efficient way to use the bypass method to produce better applications. This chapter presents results from applying this tool to a mix of commercial, open source, and proprietary web applications.

5.1 Experiment Design

The nature of web applications makes testing very complex and tedious. In general, one can follow two approaches to test web applications using the bypass method, a generic approach and a specific approach. The generic approach is to test the application with all possible fields and all possible values for all the rules defined previously in this paper. For example, assuming a text field **t** with a constraint for *maxlength="**n**"*, test cases are generated for the value of **t** with *length=0,1,...,n,n+1*. On the other hand, the specific approach is to test a field only with values produced by the rules defined above. For the example above, the text field **t** would be tested with a value of *length=0*, and with a value of *length=**n**+1*. This experiment used the specific approach, which generates fewer test cases. In fact, the foundation of bypass testing is to test values that are restricted by the client side interface; we assume that developers perform some basic testing with values that satisfy these requirements.

5.1.1 Hypothesis

Taking into consideration that modern web applications should be designed and built with reliability, usability, and security in mind, it is assumed that the applications used as subjects have been tested by their designers. The testing method of the subject developers are not often well known and quite possibly may not be well defined. However, the fact that the subject applications are in the production phase suggests that extensive testing was performed by the developers. In addition, the subjects are applications that have been used by large number of users.

In order to validate the effectiveness of bypass testing, one can suggest that this new method will reveal additional faults on the subject applications. However, due to the diverse set of subjects used in this experiment and also the fact that testing was performed in few components of each subject, the author can not statistically generalize this proposition [6, 10, 12, 18]. Instead a null hypothesis is formed:

H_0: Bypass testing of web applications will **not** expose more faults than standard testing

5.1.2 Independent Variables

There is one independent variable in this experiment, which is the method of testing web applications. Two values are used in this study:

1. Bypass method of testing web applications.

2. Industry standard testing method conducted by the developing organization.

5.1.3 Dependent Variables

This experiment is designed to study potential faults, failures, and security vulnerabilities in web applications. Therefore, the dependent variables for this study are the types of the server responses given a request submission. The main concept of the original experiment by Offutt et al. [14] will be followed and server responses will be categorized into three areas, **valid responses (V)**, **faults and failures (F)**, and **exposure (E)**. Valid responses are when invalid inputs are adequately processed by the server. Faults and failures include invalid inputs that cause abnormal server behavior. Typically, the server's automatic error handling mechanisms will catch the errors. Finally, exposure occurs when invalid input is not recognized by the server and abnormal software behavior is exposed to the users.

An important aspect of the responses evaluation is appropriateness vs. expectancy. That is, upon the submission of an invalid request, the servers response cannot be evaluated according to the result expected, since an expected response is not defined for invalid input; as least for the end user. Thus the results for invalid requests cannot be compared with some expected response. Instead, the responses are evaluated by examining the appropriateness of the handling action for the invalid input by the application.

Preliminary results showed a variety of responses that comply with the definition of the valid response. Specifically, there are test cases that produce a specific error message by the application in regards to the invalid parameter. An example is when the server would respond with a message like *"The zip code you entered is invalid"* or *"We can not process your request"* or *"Your browser sent an invalid request"*. Some other test cases produced generic messages upon receiving invalid requests. For

instance, messages such as *"Server is unavailable, try later"* or *"Server Error"* are returned. Note that this type of server error message is displayed in the context of the application interface, so it is assumed that handling took place at the application level. This type of message may be less helpful than the others, but this research did not evaluate usability. Next, there are cases where applications treat invalid requests by apparently ignoring them and processing the request, which results in output that corresponds to a valid response. Finally, some test cases produce responses that ignore the invalid request and do not process the input. For instance, a request is submitted and the server's response is the originating web page. Therefore, a breakdown of these cases is provided as subsets of the valid response definition:

- (V1) Valid response in which the server acknowledges the invalid request and provides an explicit message regarding the violation
- (V2) Valid response in which the server produces a generic error message
- (V3) Valid response in which the server apparently ignores the invalid request and produces an appropriate response
- (V4) Valid response in which the server apparently ignores the request completely

In the author's view, the responses in V3 (provided that no error occurs in the application's database or the user's session) or V1 are the desired behavior of an application in general. On the other hand, for the responses that produce generic messages (V2), one can claim that a fault was caught somewhere, but not 100% adequate handling took place. This is bases in usability factors, which are hard to

quantify; therefore, such responses are considered valid based on the fact that we can only see the result of the test in the HTML response. Yet, errors may become exposed if the back end of the application would be available.

Since there is no access to the server, source code or media (e.g. database or files), one cannot determine whether the invalid inputs cause an invalid state on the server but the response to the user is as expected or it appears appropriate. Evaluation of the validity of a response can take place only by external inspection of the output. By classifying the valid responses in the above sub-categories one can better evaluate the error handling mechanisms.

5.2 Subjects

This section provides details about the subjects,the test cases generated, the responses from the servers, and finally some confounding variables that may have affected the results.

Several criteria were used to create a pool of subject applications for this experiment. The subjects in this experiment are selected based on complexity of the application, and the ability to perform bypass testing. The complexity criterion is estimated by examining the components that interact through the HTTP requests. One characteristic of bypass testing is that no access to the application's source code is required; therefore, the complexity of a subject is determined only by external inspection. The ability to perform bypass testing is determined by the compatibility of the web interface of each application. For example, AutoBypass currently supports parsing HTML pages with some limitations on the support of JavaScript. Thus, web interfaces that use extensive scripting to do things such as substantially modify the

DOM or different technologies, such as Macromedia Flash, cannot currently be tested.

Test data were collected from various publicly available applications. The sample included open source applications as well as proprietary commercial web applications. All of the commercial application have a large user base, yet some are significantly larger than others. This particular mix of applications is composed of a variety of web applications and their responses to bypass testing. Table 5.1 lists the 16 subject applications and their 30 components.

5.3 Results

The results are presented in this section one subject application per subsection (5.3.1-5.3.16), with a summary in subsection 5.3.17. Result tables include the number of test cases created for each subject together with the type of response as described in the dependent variables of the experiment. Note that the legend provided in table 5.2 for the types of server responses is used for all the results presented.

5.3.1 ATutor Learning Content Management System

The first subject of the study is ATutor, an Open Source Web-based Learning Content Management System (LCMS) [3]. ATutor is a product of the Adaptive Technology Resource Centre of the University of Toronto. ATutor is written in PHP and includes a large nubmer of components. The component tested in this experiment is the ATalker module, which is a Web-based text-to-speech utility. From the 367 test cases, 67.9% resulted in valid responses, 0% in faults and failures, and 32.2% in exposure. Results for this module are presented in table 5.3.

The majority of the test cases that caused exposure were generated using the value

Table 5.1: Experiment Subjects

Application	Module	Subject ID	Subsection
atutor.ca	atalker	S01A	5.3.1
demo.joomla.org	index2(pol)	S02A	5.3.2
	index2(users)	S02B	
phpMyAdmin	main	S03A	5.3.3
	setTheme	S03B	
	sqlForm	S03C	
	dbStatsForm	S03D	
brainbench.com	submitRequestInfo	S04A	5.3.4
	newuser	S04B	
myspace.com	fuseaction (events)	S05A	5.3.5
	fuseaction (music)	S05B	
nytimes.com	us-markets (marketwatch.nytimes.com)	S06A	5.3.6
mutex.gmu.edu	login	S07A	5.3.7
yahoo.com	mail, notepad	S08A	5.3.8
	Mail, Compose	S08B	
	r (desktop search reminder)	S08C	
	w (weather search)	S08D	
barnesandnoble.com	xt_manage_cart	S09A	5.3.9
	booksearch/results	S09B	
www.amazon.com	item-dispatch	S10A	5.3.10
	handle-buy-box	S10B	
bankofamerica.com	ATM locator (bofa.via.infonow.net)	S11A	5.3.11
	site search	S11B	
comcast.com	Localize.ashx	S12A	5.3.12
ecost.com	detail_submit	S13A	5.3.13
	shopping_cart	S13B	
google.com	froogle	S14A	5.3.14
	language Tools	S14B	
pageflakes.com	PageFlakesRegistration	S15A	5.3.15
wellsfargolife.com	results	S16A	5.3.16

Table 5.2: Result Legend

Symbol	Response Type
V	**Valid Responses** (includes V1,V2,V3,V4)
V1	Valid response in which the server acknowledges the invalid request and provides an explicit message regarding the violation
V2	Valid response in which the server produces a generic error message
V3	Valid response in which the server apparently ignores the invalid request and produces an appropriate response
V4	Valid response in which the server apparently ignores the request completely
F	**Faults and Failures**
E	**Exposure**
T	**Total**

violation rules. Most of these test cases caused the application to reference a file that does not exist. As a result an *HTTP error 404* was presented in the main body of the interface, while the navigation, the header, and footer of the interface printed correctly. Other invalid requests created a corrupted output from the application. In this case the output is a sound file which is produced by the text-to-speech utility according to the input parameters specified in the web interface. Typically, other parameters (e.g. language setting) with invalid values were used to produce the speech file instead the "textIn" which specifies the text to become speech.

Finally, from the results of this module, one can see that more than 90% of the test cases were generated using the value violation rules. Note that as described in the previous chapters, AutoBypass does not implement the script constraint violations. The tester can inspect the HTML interface and detect fields that use scripting to detect invalid inputs and also types of controls that have predefined values. In such cases test values can be used by selecting one or more typical invalid strings provided in the AutoBypass interface. In this experiment, invalid characters are used as inputs

Table 5.3: Results for ATutor

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**224**	2	0	194	28	**0**	**110**	**334**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**24**	1	0	22	1	**0**	**8**	**32**
Total	**249**	3	0	216	30	**0**	**118**	**367**
%	**67.85%**	0.82%	0.00%	58.86%	8.17%	**0.00%**	**32.15%**	

to controls that do not normally allow modifications (e.g. hidden, radio, and checkbox controls) and thus, they are included to the value violation test cases. In this case, both the value and the field element violation rules produce similar ratios of exposures (0.33 and 0.25 respectively).

5.3.2 Joomla Content Management System

Joomla is a Content Management System (CMS) that is used by thousands of users to create and administer online portals [4]. A demo server is available for trials of the application, which was used to perform the tests on password protected pages. In Joomla, all modules of the system are accessed through a common interface. By passing parameters in GET requests, the application renders different component management interfaces through the same form and uses JavaScript to hide parts of the interface, depending on the module administered.

Poll Administration Module

First, the poll administration module was tested by generating 426 test cases. In these tests, invalid inputs replace the poll's default information (e.g. the question's

Table 5.4: Results for Joomla CMS, Poll Administration module

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**218**	0	0	5	213	**0**	**104**	**322**
Transfer Mode	**2**	0	0	0	2	**0**	**0**	**2**
Field Element Violation	**82**	0	0	1	81	**0**	**4**	**86**
Target URL Violation	**16**	0	0	0	16	**0**	**0**	**16**
Total	**318**	0	0	6	312	**0**	**108**	**426**
%	**74.65%**	0.00%	0.00%	1.41%	73.24%	**0.00%**	**25.35%**	

content, display parameters, user access, availability etc.). The majority of the tests resulted in valid responses (74.65%), and the other (25.35%) produced exposure of the application. The errors found in this module are mainly caused by parameters that are used to control the page's navigation as the application allows arbitrary inputs to be passed as part of the navigation menus. The remaining invalid responses exposed a creation of a new poll, despite the fact that the submission was indented for the update of an existing poll. In particular, by modifying the parameter 'id', which apparently corresponds to the id of the poll, the application is creating a new poll without checking if this action was legitimately requested or if the id value is within the appropriate range.

With respect to the rules used to generated test cases, 33% of value violations caused exposures, and only 9.3% of the field element violations caused exposures. Transfer mode and target URL violations produced only valid responses. Finally, in 99% of the responses that are classified as valid, it appears that the server ignores the invalid input and does not process the request (V4) since the original page was re-printed. Results for this module are presented in table 5.4.

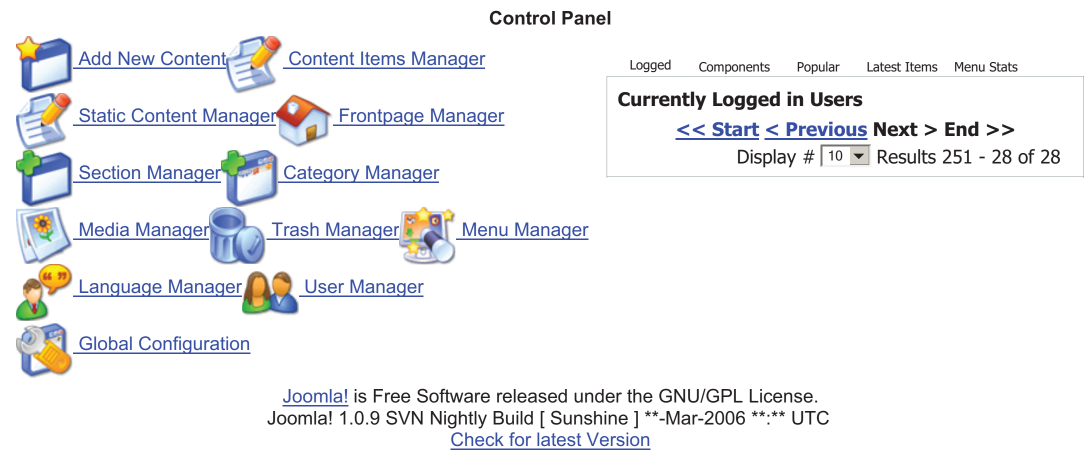


Figure 5.1: Exposure at Joomla CMS, Online User Information Module

Online User Information Module

The online user information is displayed on the main page of the CMS administrator and provides access to all users that are currently logged on. A super administrator may force a log out for a user or change the user's account settings. The tests generated resulted in 68.6% valid responses and 31.4% exposure of application errors. By modifying hidden parameters, the test requests cause an invalid argument to be passed in a control loop, causing errors on the display. In this particular installation version of the demo, script errors are output to the response and thus detectable. If the error output was disabled, this exposure could not be identified. Some other invalid requests exposed the wrong number of current users. For instance, figure 5.1 shows the application output *"display: results 251-28 of 28"* while no users are listed. Results for this module are presented in table 5.5.

Table 5.5: Results for Joomla CMS, Online User Information module

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**52**	0	0	52	0	**0**	**27**	**79**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**6**	0	0	6	0	**0**	**0**	**6**
Total	**59**	0	0	59	0	**0**	**27**	**86**
%	**68.60%**	0.00%	0.00%	68.60%	0.00%	**0.00%**	**31.40%**	

5.3.3 PhpMyAdmin, Web Based MySQL Client

The phpMyAdmin web application allows users to administer MySQL [5] databases. According to the project statistics, there have been approximately eight million downloads from the launch of the project until the middle of December in 2005. PhpMyAdmin is tested using a demo server that is available by the developers.

Current Database Control

The first element of phpMyAdmin that was tested is the form that sets the current database. In this first test set, all responses are valid. Yet, it is noticeable that the majority of the test cases (82%) result in a valid response. In these cases the server apparently ignored invalid inputs and did not process the request. The remaining 18% are caught by the application and explicitly acknowledged. Results for this module are presented in table 5.6.

Theme Control

Next, the form that sets the theme of the HTML interface was tested. In this set, faults in the validations are exposed in almost 15% of the invalid requests. In fact,

Table 5.6: Results for PhpMyAdmin, Current Database Control

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**45**	10	0	0	35	**0**	**0**	**45**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**8**	0	0	0	8	**0**	**0**	**8**
Total	**54**	10	0	0	44	**0**	**0**	**54**
%	**100.00%**	18.52%	0.00%	0.00%	81.48%	**0.00%**	**0.00%**	

Table 5.7: Results for PhpMyAdmin, Set Theme Module

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**76**	0	0	0	76	**0**	**15**	**91**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**10**	1	0	0	9	**0**	**0**	**10**
Total	**87**	1	0	0	86	**0**	**15**	**102**
%	**85.29%**	0.98%	0.00%	0.00%	84.31%	**0.00%**	**14.71%**	

the application allows a call to a non-existing style sheet through a form that consists of hidden controls selection menus. The demo installation allows the error messages to print to the output, giving the author the ability to identify the exposure. For this component, all faults are exposed by the value violation rules with a rate of 15 errors on 91 test cases (16.5%). Results for this module are presented in table 5.7.

SQL Query Form

The SQL query input form resulted in no failure or exposure. All test cases produces valid responses as the invalid requests are either ignored (98.5%) or explicitly acknowledged (1.5%). Results for this module are presented in table 5.8.

Table 5.8: Results for PhpMyAdmin, SQL Form

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**96**	2	0	0	94	0	0	**96**
Transfer Mode	**1**	0	0	0	1	0	0	**1**
Field Element Violation	**36**	0	0	0	36	0	0	**36**
Total	**133**	2	0	0	131	**0**	**0**	**133**
%	**100.00%**	1.50%	0.00%	0.00%	98.50%	**0.00%**	**0.00%**	

Table 5.9: Results for PhpMyAdmin, Data Base Statistics

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**108**	0	0	108	0	**0**	**0**	**108**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**17**	0	0	16	1	**0**	**0**	**17**
Total	**126**	0	0	125	1	**0**	**0**	**126**
%	**100.00%**	0.00%	0.00%	99.21%	0.79%	**0.00%**	**0.00%**	

SQL Database Statistics Form

The form used to produce status reports for the databases was parsed to generated test cases. This module also returned valid responses. Results for this module are presented in table 5.9.

5.3.4 Brainbench.com Online Assessment Products

The first commercial application tested in this experiment is `brainbench.com`. An online assessment provider with impressive clientele and achievements in the industry, `brainbench.com` is a portal with a significant number of users.

Table 5.10: Results for `Brainbench.com`, Information Request Form

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Violation	**0**	0	0	0	0	**9**	**0**	**9**
Value Violation and Invalid Characters	**0**	0	0	0	0	**103**	**0**	**103**
Transfer Mode	**0**	0	0	0	0	**1**	**0**	**1**
Field Element Violation	**0**	0	0	0	0	**37**	**1**	**38**
Total	**0**	0	0	0	0	**150**	**1**	**151**
%	**0.00%**	0.00%	0.00%	0.00%	0.00%	**99.34%**	**0.66%**	

Business Information Request Form

The application is heavily dependent on the client side validation for business information request module. In all but one test case, the server software exception mechanism was activated. The other test case exposed failure when an invalid set of values was submitted. Results for this module are presented in table 5.10.

New User Registration

The registration module produced an interesting variety of responses. To business requirements, the validity of the user information is crucial for this application. All certifications and assessment ratings for the users will be assigned to the account created in this form. The test set generated for this component included 281 invalid requests. In fact, the AutoBypass automated submission mechanism did not produce correct results since all test cases use the same login ID, and the application directly identifies this request as trying to use the login ID of an existing account. Instead, manual submission of mutants allowed the tester to modify the request to produce a correct test. For this set, the email value was altered manually in each mutated form.

Even though the application handles the majority (80.8%) of the invalid request with valid responses. From these, 28.11% identified most invalid inputs explicitly and a message produced a warning about the input requirements and 52.7%apparently ignored parameters that are most likely used for internal usage (i.e. not parameters for the user account like name, password, etc.) and allowed the registration. The remaining (19.2%) exposes various errors. The author is not in a position to examine the effects of these tests since the server is not available. Errors typically include cases where invalid new user requests are allowed, despite the invalid inputs on key parameters for the user registration (e.g. empty password confirmation, address, state, etc.).

Interestingly, some invalid parameter values had unexpected effects on the messages produced by the application. For instance, the violation of the *maxlength* of the address parameters caused a warning for an existing account. After limiting the length of the address value, the account creation is allowed with the same login ID; therefore, these cases are classified as errors. The field element and value violation rules produced a similar percentage of fault exposures (17.4% and 18.3% respectively) and half of the test cases generated by length violation rules produced exposure errors. Results for this module are presented in table 5.11.

5.3.5 Myspace.com Online Community Portal

`Myspace.com` is a popular portal for a young audience that was initially focused in music events. This portal is now one of the most popular community portals for users to create accounts that are used to share photos and journals and also communicate with other members.

Table 5.11: Results for `Brainbench.com`, New User Registration

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Violation	**5**	5	0	0	0	**0**	**5**	**10**
Value Violation and Invalid Characters	**183**	59	0	124	0	**0**	**41**	**224**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**38**	15	0	23	0	**0**	**8**	**46**
Total	**227**	79	0	148	0	**0**	**54**	**281**
%	**80.78%**	28.11%	0.00%	52.67%	0.00%	**0.00%**	**19.22%**	

Table 5.12: Results for `Myspace.com`, Event Search

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Violation	**2**	0	0	2	0	**0**	**0**	**2**
Value Violation and Invalid Characters	**54**	0	0	54	0	**0**	**0**	**54**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**20**	0	0	20	0	**0**	**0**	**20**
Total	**77**	0	0	77	0	**0**	**0**	**77**
%	**100.00%**	0.00%	0.00%	100.00%	0.00%	**0.00%**	**0.00%**	

Event Search

The first test set for this application was generated around the event search form. None of the test cases produced faults or exposure. All of the invalid requests are apparently ignored by the application, which produces typical responses (i.e. provide results for the search requested). The results for this module are presented in table 5.12.

Table 5.13: Results for `Myspace.com`, Music Search

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**49**	0	26	23	0	**0**	**2**	**51**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**10**	0	1	9	0	**0**	**0**	**10**
Total	**60**	0	27	33	0	**0**	**2**	**62**
%	**96.77%**	0.00%	43.55%	53.23%	0.00%	**0.00%**	**3.23%**	

Music Search

Next, the interface that is used to search a different section of the content was tested. Most of these test cases produced valid responses (96.8%); yet, a large percentage (43.6%) of the generated requests caused the application to produce generic messages. In particular, the error message displayed was "*Sorry! an unexpected error has occurred*". This is considered a valid response since the application acknowledges some form of invalid input and prevents an exposure or crash. However, we cannot be sure whether invalid inputs affected the database or other server-side component of the application. 3.23% of the test cases generated produced exposure, all of which were produced by value violation rules. By modifying the value of the "page" parameter, the results returned were incorrectly identified by the page numbering. Results for this module are presented in table 5.13.

5.3.6 NYtimes.com Online New

The next subject is the New York Times online edition. The interface used for test generation was in the business section (`marketwatch.nytimes.com`), which was built using the Active Server Pages (ASP) technology. This component provides an

overview of the stock markets and allows users to search for specific trade symbols. The invalid requests resulted in 55.4% valid responses, 1.42% faults and failures, and 43.18% exposures.

Most valid responses appear to ignore the invalid request parameters and produce an appropriate response. In a few cases (1.1%), a generic message was received. Upon invalidating the parameter "guid", the application responded with a generic message ("*story not available*") in the body of the page. Faults and failures were caused by 1.4% of the test cases; the application did not respond at all. The author cannot be sure of the reason of such reaction, but the use of advertising mechanisms is suspected. In either case, the application does not respond with any acknowledgement.

Finally, 43.2% of the invalid requests exposed faults. All of the test cases that resulted in exposure were generated using the value violation rules. By altering values in hidden input controls, the application referenced images that do not exist in place of the graphs for each market (i.e. an empty image is rendered). A second type of error is the violation of specific parameters, namely "screen" and "exchange", resulted in the omission of the last part of the output. Initially, it appeared that an HTML tag was not output correctly but part of the output was replaced by the ASP error report. Results for this module are presented in table 5.14.

5.3.7 Mutex.gmu.edu Libraries Database Gateway

The George Mason University Libraries Database Gateway application is used to authenticate students for access to the university's subscriptions to online periodicals and journals. The Mutex application is the proxy gateway for accessing these resources. The interface that is used by students to input their ID is very simple,

Table 5.14: Results for `NYtimes.com`, Market Watch

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**40**	0	0	39	1	2	147	**189**
Transfer Mode	**1**	0	0	1	0	0	0	**1**
Field Element Violation	**9**	0	0	9	0	0	5	**14**
Target URL Violation	**145**	0	4	0	141	3	0	**148**
Total	**195**	0	4	49	142	**5**	**152**	**352**
%	**55.40%**	0.00%	1.14%	13.92%	40.34%	**1.42%**	**43.18%**	

Table 5.15: Results for `mutex.gmu.edu`, University Libraries Database Gateway

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**28**	28	0	0	0	**0**	**54**	**82**
Transfer Mode	**1**	1	0	0	0	**0**	**0**	**1**
Field Element Violation	**1**	1	0	0	0	**0**	**7**	**8**
Total	**30**	30	0	0	0	**0**	**61**	**91**
%	**32.97%**	32.97%	0.00%	0.00%	0.00%	**0.00%**	**67.03%**	

only one text input control is rendered. However, the HTML source code contains a numbers of hidden fields that carry information used by the application to determine the target journal to be accessed. A test set was generated by modifying these parameters, resulting in 33% valid responses. All of the valid responses are explicitly acknowledged by the application with the message: "*That number was not recognized. Try again below*". The remaining 67% of the test cases generated responses that exposed errors handled by the application's server. The request was redirected to nonexisting pages, causing the server to sent an "*Error 404–Not Found*" to the browser. Results for this module are presented in table 5.15.

Table 5.16: Results for Yahoo.com, mail: notepad

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**65**	0	0	0	65	**0**	**0**	**65**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**32**	0	0	0	32	**0**	**0**	**32**
Total	**98**	0	0	0	98	**0**	**0**	**98**
%	**100.0%**	0.0%	0.0%	0.0%	100.0%	**0.0%**	**0.0%**	

5.3.8 Yahoo.com, Global Internet Service

Yahoo.com allows users to access an array of services (search, personalized pages, email, news and many others) through the web. Yahoo is selected in this experiment's sample to represent some of the applications with the largest volume of transactions in the market. User authentication was required to test some components. The AutoBypass' authentication mechanism was used for that purpose, and the author's personal account was provided for login.

Notepad

The first component tested was notepad, which is one of the many features provided in the overall web email application. The interface used to test this module was the Edit Note form. All of the 98 generated tests produced valid responses that appear to ignore the violated parameters and processed the requests with no evidence of errors. Results for this module are presented in table 5.16.

Table 5.17: Results for `Yahoo.com`, compose message

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**314**	0	0	0	314	**0**	**0**	**314**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**155**	0	0	0	155	**0**	**0**	**155**
Total	**470**	0	0	0	470	**0**	**0**	**470**
%	**100.0%**	0.0%	0.0%	0.0%	100.0%	**0.0%**	**0.0%**	

Message Composer

Composing email messages is one of the most common actions of yahoo's email users. The interface constructed for this function was used to generate the next test set. All generated test cases resulted in valid responses, consistent with the results of the previous component of this application. Results for this module are presented in table 5.17.

Yahoo Toolbar Reminder

The entry point of the Yahoo web site is often used to advertise new services. Depending on the history of a user, a notification of the browser search extension is displayed on top of the web page. The user may click on a check box to disable the appearance of this notification in future visits. Surprisingly, 99% of the 37 test cases for this small form resulted in a faults and failures. The application returned an empty screen that includes a string of character that was not rendered on the browser. The only test cases that produced a valid response were generated by the transfer mode violation. By changing the form method from GET to POST, the application returned the message that this method is not allowed. The results for this module

Table 5.18: Results for `Yahoo.com`, Desktop Search Reminder

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**0**	0	0	0	0	**31**	**0**	**31**
Transfer Mode	**1**	1	0	0	0	**0**	**0**	**1**
Field Element Violation	**0**	0	0	0	0	**5**	**0**	**5**
Total	**1**	1	0	0	0	**36**	**0**	**37**
%	**2.70%**	2.70%	0.00%	0.00%	0.00%	**97.30%**	**0.00%**	

Table 5.19: Results for `Yahoo.com`, Weather and Traffic Search

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**54**	0	0	54	0	**0**	**0**	**54**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**4**	0	0	4	0	**0**	**0**	**4**
Total	**59**	0	0	59	0	**0**	**0**	**59**
%	**100.0%**	0.0%	0.0%	100.0%	0.0%	**0.0%**	**0.0%**	

are presented in table 5.18.

Weather and Traffic Search

The last component tested in this application was the weather and traffic search tools, which are also part of the main yahoo page. Using a location name or zip code, one can find details about the weather and traffic conditions. All invalid requests generated for this component's interface produce appropriate results. Results for this module are presented in table 5.19.

Table 5.20: Results for Barnesandnoble.com, Shopping Cart

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**123**	0	1	122	0	**0**	**2**	**125**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**10**	0	1	9	0	**0**	**0**	**10**
Target URL Violation	**40**	0	5	35	0	**0**	**0**	**40**
Total	**174**	0	7	167	0	**0**	**2**	**176**
%	**98.86%**	0.00%	3.98%	94.89%	0.00%	**0.00%**	**1.14%**	

5.3.9 Barnesandnoble.com, Online Store

Continuing with commercial applications with a high volume of transactions, the barnsandnoble.com online store was tested. With a long retail history, the book store chain expanded their online operations to other retail markets such as music and video. Only public sections of the web application were tested; no authentication was required to access these pages. Two components from this subject are tested and the results follow.

Shopping Cart

A critical function of online stores is the shopping cart component. As each product detail is displayed on the application's interface, the "add to shopping cart" form allows a user to add the current item to the collection that he or she wants to purchase. Specifically in the video collection, AutoBypass generated 176 test cases from which 98.9% resulted in valid responses and 1.1% in exposure of errors. Results for this module are presented in table 5.20.

Valid responses mainly consisted of output that suggests that the application does

not take into account the invalid parameters; the output was valid responses. Yet some of the valid responses resulted in generic error messages. Two specific messages, are:

Message A:

> *We are sorry... Our system is temporarily unavailable due to routine maintenance. We apologize for any inconvenience during this outage and expect our systems to return shortly. Thank you for your patience.*

Message B:

> *Sorry, we cannot process your request at this time.*
> *We may be experiencing technical difficulties, or you may need to adjust your browser settings. If you continue to have problems, please click here to send a message to customer service.*

By examining the output captured by AutoBypass, test cases that generated message B were initially classified as invalid responses. During the test, the application responded with an empty text file with the phrase:"*The parameter is incorrect*". However, as the results were manually verified using a regular browser, the application produced an appropriate page (i.e. message B within the regular header and footer of the application's interface). In fact, web applications may produce different responses for different clients; e.g. a desktop browser vs. a browser in a hand-held device. It is assumed that Autobypass was not recognized as one of the typical web browsers and a simplistic error message was generated.

On the other hand, the small percentage of test cases that cause exposures are due to invalid inputs that violate the cart display function. Even through an error

Table 5.21: Results for `Barnesandnoble.com`, Book Search

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**132**	13	0	119	0	**3**	**0**	**135**
Transfer Mode	**1**	1	0	0	0	**0**	**0**	**1**
Field Element Violation	**6**	0	0	6	0	**0**	**0**	**6**
Total	**139**	14	0	125	0	**3**	**0**	**142**
%	**97.89%**	9.86%	0.00%	88.03%	0.00%	**2.11%**	**0.00%**	

message was received explaining that the item could not be added to the shopping cart, the page also displays an empty cart with the title "*You just added this item to your Cart*". In addition, the cart display uses JavaScipt, which due to the lack of references in the card prints undefined values (e.g. NaN). Further, all other sections of the pages (such as wish list, suggested items, and special offers) were empty.

Book Search

The second component tested from this subject was the book search tool. Users may perform a quick search for a book listing using this interface. The results of the tests generated using this form were primarily valid responses (97.89%), while the remaining ones (2.11%) produced faults and failures. The majority of the invalid requests (88%) generated valid responses that appear to ignore the invalid inputs and process the request with no apparent error. A smaller fraction of the test cases (9.9%) produced valid responses with explicit messages regarding the invalid requests. Finally, three failures are detected when the hidden input control named "TXT" is violated with a string that starts with a percent sign. An example of such a failure is depicted in figure 5.2. Results for this module are presented in table 5.21.

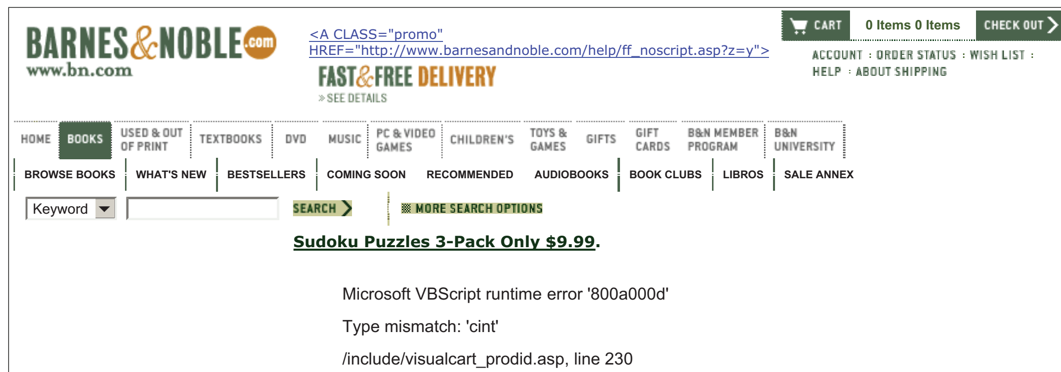


Figure 5.2: Fault/Failure at `Barnesandnoble.com`, Book Search component

Table 5.22: Results for `Amazon.com`, Item Dispatcher

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**52**	0	0	0	52	0	0	**52**
Transfer Mode	**1**	0	0	0	1	0	0	**1**
Field Element Violation	**10**	0	0	0	10	0	0	**10**
Total	**63**	0	0	0	63	**0**	**0**	**63**
%	**100.00%**	0.00%	0.00%	0.00%	100.00%	**0.00%**	**0.00%**	

5.3.10 Amazon.com, Online Store

Amazon in one of the largest online stores in the market and a leader in web technologies. Two interfaces found at the `amazon.com` site were tested and the results are the following:

Item dispatcher

First, the item dispatcher component is the mechanism to control the shopping cart of a user. All test cases produced 100% valid responses. Results for this module are presented in table 5.22.

Table 5.23: Results for `Amazon.com`, Handle Buy

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	1	0	0	1	0	0	0	1
Transfer Mode	1	0	0	1	0	0	0	1
Field Element Violation	31	0	0	31	0	0	0	31
Total	33	0	0	33	0	0	0	33
%	100%	0.00%	0.00%	100%	0.00%	0.00%	0.00%	

Handle Buy (Shopping Cart)

The second component that was tested, handle buy, also produced 100% valid responses. Essentially, this component is the equivalent of a shopping cart controller typically found in online stores. One interesting outcome from this test was the incorrect resolution of the relative paths for the URL of the form submission for the handle buy component. The complex URLs used by this application caused AutoBypass to append the relative path of the form action to the document's URL including the parameters. That resulted in forms that submited requests to a URL of a form: `domain/path1/path2/`. All results captured by AutoBypass received an empty page as a response, which originally classified the responses as invalid. One could possibly argue that this is still an incorrect behavior of the application as it should have warned the user that the path is invalid. However, the confirmation of the responses using the mutant forms and manually correcting the URL of the form action produced appropriate responses. Results for this module are presented in table 5.23.

5.3.11 Bankofamerica.com, Online Banking

One of the first to introduce online banking, Bank of America has developed a remarkable portal. Even though AutoBypass provides the ability to test password protected pages and potentially can be used to assess the correctness of online banking features, only public components of the applications that provide information were tested to assess the robustness of these applications. During previous use of AutoBypass in small applications, severe corruption of data was detected. For that reason, the author did not test modules that deal with bank transactions. Such tests would require careful preparation and possibly cooperation with the institutions to avoid unexpected results (such as financial losses) that may be caused by this experiment.

ATM & Branch Locator

The Automated Teller Machine (ATM) and Branch locator tools are part of the main page in the subject's portal. Behind the simple appearance of the interface used to access this feature, one can identify an array of hidden input controls that are used to perform the call to the application. By examining the interface's HTML source, it is also revealed that the form action URL points to a separate domain. Specifically, `bofa.via.infonow.net` hosts the application that returns the results of this action. By submitting a request, infonow.net acts as an extension of the Bank of America's portal, presenting results of the search using the bank's identity. This is not apparent to a user unless she or he carefully reads the URL on a browser. For that reason, the results are listed as part of this subject.

The application correctly handled 66% of the test cases, providing valid responses and explicitly acknowledging the invalid input in 4% of the tests. Invalid parameters

Table 5.24: `Bankofamerica.com`, ATM & Branch Locator by `infonow.net`

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Violation	**1**	1	0	0	0	**0**	**0**	**1**
Value Violation and Invalid Characters	**66**	3	0	63	0	**17**	**25**	**108**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**17**	1	0	16	0	**1**	**0**	**18**
Total	**85**	5	0	80	0	**18**	**25**	**128**
%	**66.41%**	3.91%	0.00%	62.50%	0.00%	**14.06%**	**19.53%**	

are apparently ignored 62% of the times and produce appropriate responses in which invalid requests are ignored and default locations are presented. Specifically, locations in California (headquarters of the bank) are shown. The application server handled exceptions generated by invalid inputs in 14% of the test cases. For instance, the error : "*[ServletException in:/jsp/layouts/ApplicationTemplate.jsp] Error - Tag Insert : No value found for attribute* ***metaText****.*" is returned from one of these cases. Finally, the remaining 20% of the test cases exposed errors. Typically the invalid requests caused the application to produce responses referring to the default locations (CA) while the originating request included a zip code for Virginia. In addition, the links provided for driving directions are incorrect. An example is shown in figure 5.3. The value violation rules produced the majority of the invalid responses (99.9%) and only one test generated invalid responses using field element violation rules. Results for this module are presented in table 5.24.

Site Search

The second module is the site search tool in the main portal page. The requests are handled by the same application (not as in the previous component). All of the

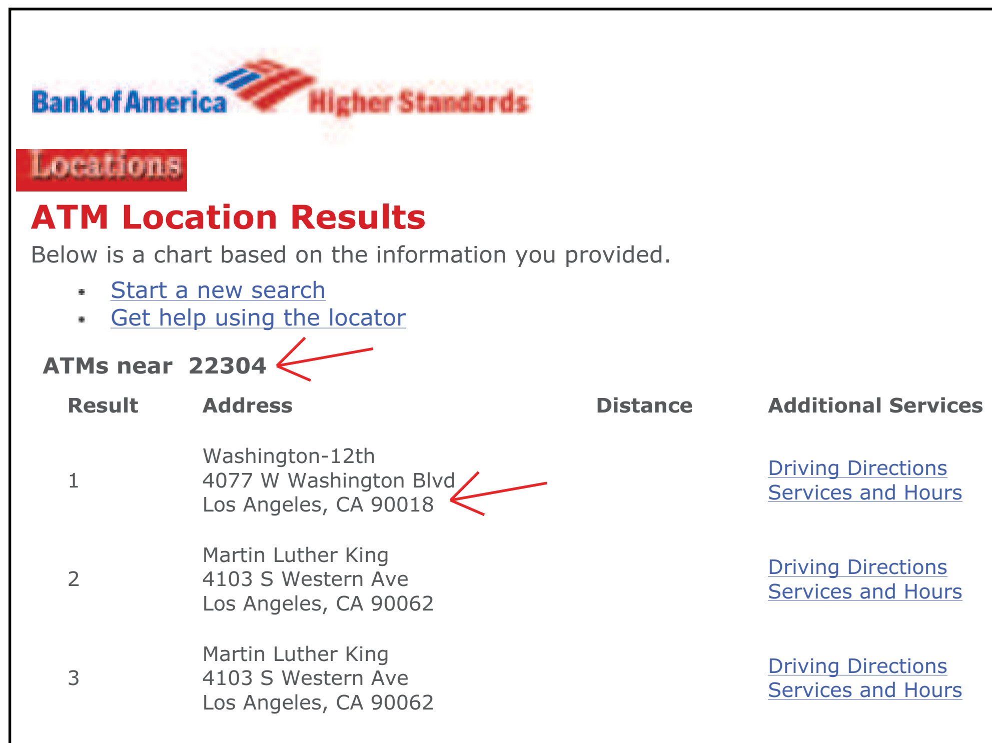


Figure 5.3: Exposure at `Bankofamerica.com`, ATM & Branch Locator

Table 5.25: `Bankofamerica.com`, Web Site Search

Violation Type	Server Response							Total
	V	**V1**	**V2**	**V3**	**V4**	**F**	**E**	
Length Violation	**1**	0	0	0	1	**0**	**0**	**1**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**4**	0	0	0	4	**0**	**0**	**4**
Target URL Violation	**204**	0	2	142	60	**0**	**0**	**204**
Total	**210**	0	2	142	66	**0**	**0**	**210**
%	**100%**	0.00%	0.95%	67.62%	31.43%	**0.00%**	**0.00%**	

invalid requests produce appropriate responses. For the majority (68%) of the test cases, the application ignored the invalid requests. The server ignored the request for 31% of the cases and responded with the original interface. Finally, two tests (1%) produce generic messages (e.g. *"The Bank of America page you are trying to reach: http://www.bankofamerica.com/stateerror is not available"*). Results for this module are presented in table 5.25.

5.3.12 Comcast.net, Communications Provider

A typical functionality in web sites for communications providers is the service availably indicators. Users interested in a particular service can verify the availability for a location. The interface provided in `comcast.net` was used to generate 105 test cases, all of which produced valid responses. Yet, only 15% explicitly acknowledge the invalid parameters, 82% produce generic messages (e.g. *"We apologize. The area of Comcast.com you're attempting to access is temporarily unavailable"*), and 3% ignore the invalid request. Results for this module are presented in table 5.26.

Table 5.26: `Comcast.net`, Service Availability

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Violation	**3**	0	3	0	0	**0**	**0**	**3**
Value Violation and Invalid Characters	**80**	16	63	0	1	**0**	**0**	**80**
Transfer Mode	**1**	0	1	0	0	**0**	**0**	**1**
Field Element Violation	**20**	0	18	0	2	**0**	**1**	**21**
Total	**104**	16	85	0	3	**0**	**1**	**105**
%	**99%**	15.24%	80.95%	0.00%	2.86%	**0.00%**	**0.95%**	

5.3.13 Ecost.com, Online Store

Ecost is another online store. Two typical interfaces in online stores are the product details and the shopping card controller with which users specify items to purchase.

Product Detail

Test cases generated for the product details component revealed inappropriate handling in 73% of the cases. In particular, 41 cases produced failures and 5 exposed application errors. The remaining 27% of the generated invalid requests were handled appropriately. Overall, value violation rules were used to generate 55 requests; 81% of these produced invalid responses. On the other hand, field element violation rules generated 21 cases from which 28.5% resulted in failures and errors. Results for this module are presented in tables 5.27.

Shopping Cart

The next component in this experiment is the shopping cart controller for `ecost.com`. Most test cases (61%) resulted in valid responses and 39% produced invalid responses. Specifically, faults were caught by the ASP runtime engine. Results for this module

Table 5.27: `Ecost.com`, Shopping Cart

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**10**	0	0	9	1	**41**	**4**	**55**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**6**	0	0	6	0	**0**	**1**	**7**
Total	**17**	0	0	15	2	**41**	**5**	**63**
%	**26.98%**	0.00%	0.00%	23.81%	3.17%	**65.08%**	**7.94%**	

Table 5.28: `Ecost.com`, Shopping Cart

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**82**	0	0	82	0	**56**	**0**	**138**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**15**	0	0	14	1	**6**	**0**	**21**
Total	**98**	0	0	97	1	**62**	**0**	**160**
%	**61.25%**	0.00%	0.00%	60.63%	0.63%	**38.75%**	**0.00%**	

are presented in table 5.28.

5.3.14 Google.com, Search Engine

Google is one of the giants in the web technologies and services. Google's pages are accessed by vast amounts of users throughout the globe. AutoBypass generated 148 test cases for the first component, the shopping search engine Froogle, one of Google's services. Next, the language tool feature was tested, which allows users to search pages in specific languages. In both sets, 100% of responses are valid. Results for these modules are presented in table 5.29 and table 5.30 respectively.

Table 5.29: `Google.com`, Froogle - Shopping Search Engine

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**136**	0	0	136	0	**0**	**0**	**136**
Transfer Mode	**1**	0	0	0	1	**0**	**0**	**1**
Field Element Violation	**11**	0	0	10	1	**0**	**0**	**11**
Total	**148**	0	0	146	2	**0**	**0**	**148**
%	**100.00%**	0.00%	0.00%	98.65%	1.35%	**0.00%**	**0.00%**	

Table 5.30: `Google.com`, Language Tools

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**94**	0	0	94	0	**0**	**0**	**94**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**12**	0	0	12	0	**0**	**0**	**12**
Total	**107**	0	0	107	0	**0**	**0**	**107**
%	**100%**	0.00%	0.00%	100%	0.00%	**0.00%**	**0.00%**	

Table 5.31: `Pageflakes.com`, User Registration

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Value Violation and Invalid Characters	**130**	69	0	0	61	**48**	**0**	**178**
Transfer Mode	**1**	1	0	0	0	**0**	**0**	**1**
Field Element Violation	**19**	18	0	0	1	**3**	**0**	**22**
Total	**150**	88	0	0	62	**51**	**0**	**201**
%	**74.63%**	43.78%	0.00%	0%	30.85%	**25.37%**	**0.00%**	

5.3.15 Pageflakes.com, Community Personalized Portal

Pageflakes is a new community portal that is built on a relatively new approach for developing rich web interfaces, Asynchronous JavaScript and XML (AJAX) [1]. The user registration component was selected to generate tests. The majority (75%) of the test cases produced valid responses. Responses that explicitly acknowledged the invalid input are 44% of the overall test cases. Responses that ignore the request were 31% of the overall test cases. However, 25% of the invalid requests produced faults handled by the web server. Results for this module are presented in table 5.31.

5.3.16 Wellsfargolife.com, Life Insurance

The last subject in this experiment is the Wells Fargo Insurance portal. The form used to generate life insurance quotes was tested. The invalid requests generated by AutoBypass resulted in significant percentage of faults (70%) and the remaining produced valid responses. Most of the test that produced invalid responses were created using value violation rules. Results for this module are presented in table 5.32.

Table 5.32: `Wellsfargolife.com`, Insurance Quote

Violation Type	Server Response							Total
	V	V1	V2	V3	V4	F	E	
Length Violation	**0**	0	0	0	0	**1**	**0**	**1**
Value Violation and Invalid Characters	**69**	0	13	54	2	**210**	**0**	**279**
Transfer Mode	**1**	0	0	1	0	**0**	**0**	**1**
Field Element Violation	**22**	0	6	16	0	**1**	**0**	**23**
Total	**92**	0	19	71	2	**212**	**0**	**304**
%	**30.26%**	0.00%	6.25%	23%	0.66%	**69.74%**	**0.00%**	

5.3.17 Result Summary

A summary of the results provides some information about the overall performance of the automated tool and the bypass methodology for testing web applications. A summary of the results for the experiment is in table 5.33. It can be seen from the data that an impressive 24% of the test cases were inappropriately handled by the web applications used in this experiment. Specifically 12% of the test cases generated faults and failures and 12% exposure. Table 5.34 shows the various types of invalid responses from the subjects.

Since that subjects are applications in production mode, the standard testing by their developers is assumed to be finalized. Thus, Hypothesis H_0 stated that bypass testing will **not** expose more faults than standard testing. Tests on the `amazon.com` (S10) and `google.com` (S14) applications produced zero invalid responses (faults or exposures). Therefore, hypothesis H_0 is verified for the above subject applications. However, the hypothesis H_0 is rejected for the remaining experiment samples and the sample population in general.

Figure 5.4 graph B (Response Types), shows the response types for each subject.

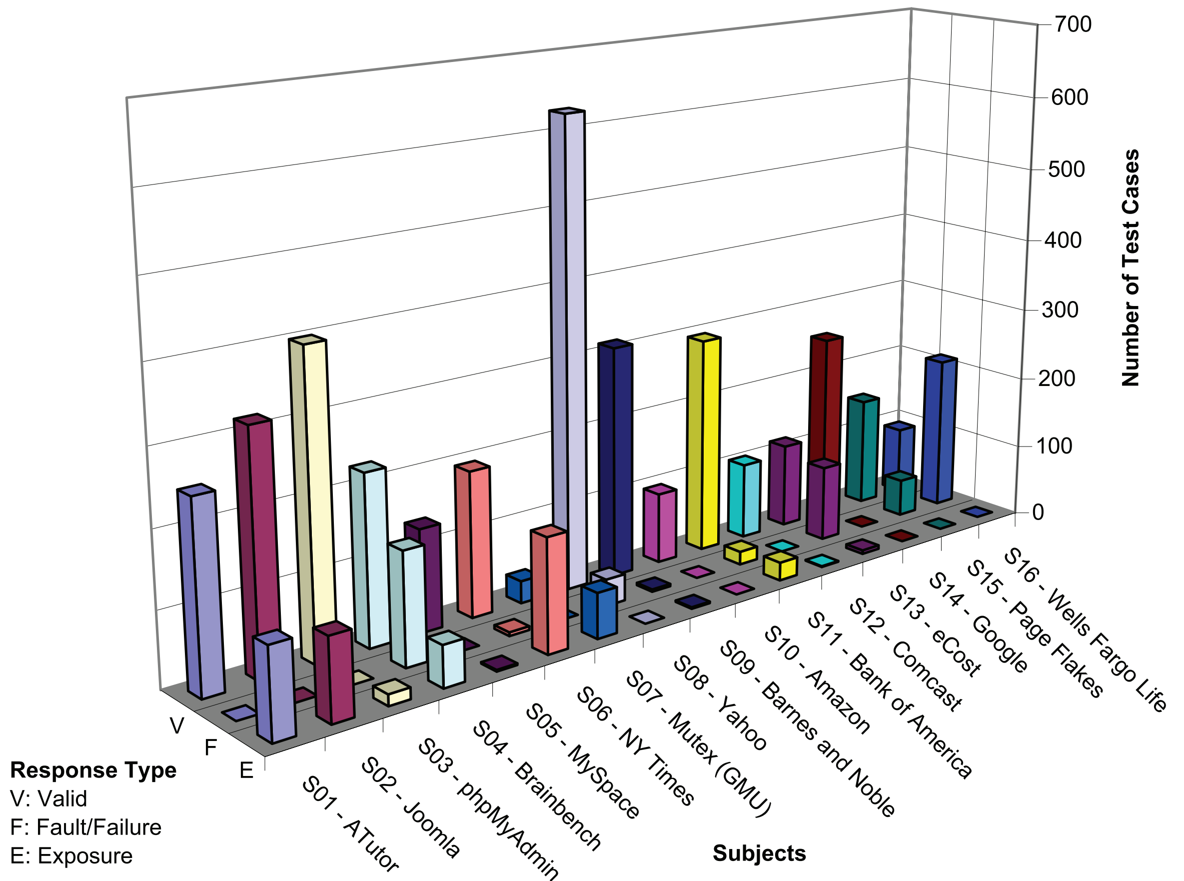
Table 5.33: Result Summary

Application	Subject ID	V	V1	V2	V3	V4	F	E	T
atutor.ca	S01A	**249**	3	0	216	30	**0**	**118**	367
demo.joomla.org	S02A	**318**	0	0	6	312	**0**	**108**	426
	S02B	**59**	0	0	59	0	**0**	**27**	86
phpMyAdmin	S03A	**54**	10	0	0	44	**0**	**0**	54
	S03B	**87**	1	0	0	86	**0**	**15**	102
	S03C	**133**	2	0	0	131	**0**	**0**	133
	S03D	**126**	0	0	125	1	**0**	**0**	126
brainbench.com	S04A	**0**	0	0	0	0	**150**	**1**	151
	S04B	**227**	79	0	148	0	**0**	**54**	281
myspace.com	S05A	**77**	0	0	77	0	**0**	**0**	77
	S05B	**60**	0	27	33	0	**0**	**2**	62
nytimes.com	S06A	**195**	0	4	49	142	**5**	**152**	352
mutex.gmu.edu	S07A	**30**	30	0	0	0	**0**	**61**	91
yahoo.com	S08A	**98**	0	0	0	98	**0**	**0**	98
	S08B	**470**	0	0	0	470	**0**	**0**	470
	S08C	**1**	1	0	0	0	**36**	**0**	37
	S08D	**59**	0	0	59	0	**0**	**0**	59
barnesandnoble.com	S09A	**174**	0	7	167	0	**0**	**2**	176
	S09B	**139**	14	0	125	0	**3**	**0**	142
amazon.com	S10A	**63**	0	0	0	63	**0**	**0**	63
	S10B	**33**	0	0	0	33	**0**	**0**	33
bankofamerica.com	S11A	**85**	5	0	80	0	**18**	**25**	128
	S11B	**210**	0	2	142	66	**0**	**0**	210
comcast.com	S12A	**104**	16	85	0	3	**0**	**1**	105
S01 - ATutor	S13A	**17**	0	0	15	2	**41**	**5**	63
	S13B	**98**	0	0	97	1	**62**	**0**	160
S01 - ATutor	S14A	**148**	0	0	146	2	**0**	**0**	148
	S14B	**107**	0	0	107	0	**0**	**0**	107
pageflakes.com	S15A	**150**	88	0	0	62	**51**	**0**	201
wellsfargolife.com	S16A	**92**	0	19	71	2	**212**	**0**	304
total tests		**3663**	249	144	1722	1548	**578**	**571**	**4812**
%		**76.1%**	5.2%	3.0%	35.8%	32.2%	**12.0%**	**11.9%**	

Legend	
V	**Valid response (includes V1,V2,V3,V4)**
V1	Valid response with server producing explicit error message
V2	Valid response with server producing generic error message
V3	Valid response with server ignoring invalid request
V4	Valid response with server ignoring and not processing request
F	**Faults and Failures**
E	**Exposure**
T	**Total**

Table 5.34: Types of Invalid Responses

Application	Types of Invalid Responses
Atutor	Reference of non existing files within main frame of interface Corrupted output (sound files) Incorrect parameter reference for output
Joomla	Invalid characters appeared in navigation bar New polls created upon invalid parameters on existing polls Current user display was incorrect Invalid character caused errors caught by runtime engine
phpMyAdmin	Invalid character caused errors caught by runtime engine Invalid input (character set) reached database queries
Brainbench	Exceptions caught by web server (request info) Request of information were submitted with bogus data Invalid new user requests were allowed Violation of specific parameters produced irrelevant errors
Myspace	Page number on results was incorrect
NYtimes	Application referenced images that did not exist Parts of the output where omitted VB runtime errors
Mutex.gmu.edu	Invalid requests were redirected to nonexistent pages
Yahoo	Invalid request resulted to empty response (desktop search reminder)
Barnes & Nobles	Incorrect output VB runtime errors
Amazon	NONE
Bank of America	Exceptions caught by web server (ATM search) Incorrect results (ATM search)
Comcast	Incorrect result
Ecost	VB runtime errors
Google	NONE
Pageflakes	.NET runtime errors
Wellsfargolife	VB runtime errors



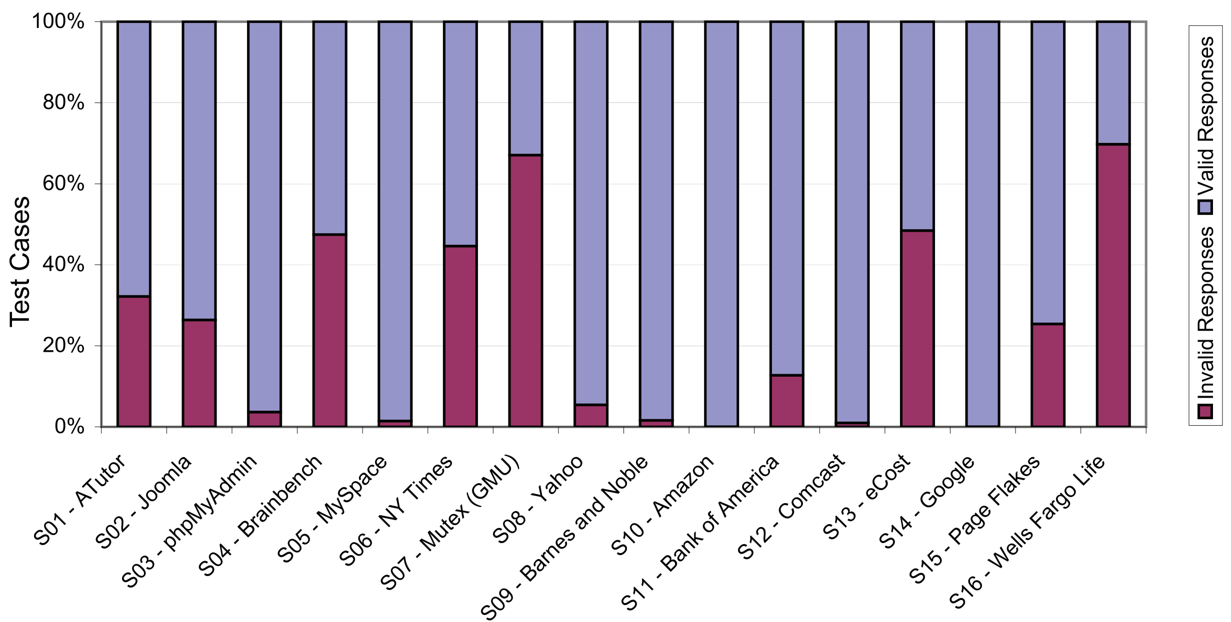


Figure 5.4: Result Summary Graphs

Different numbers of test cases were generated for each subject. The main factor for this variation is the combination of the rules applied to each interface and the user input. The number of test cases is directly proportional to the numbers of input controls parsed in each interface. Moreover, the tester's input to the automation tool also affect the number of test cases. Depending on the interface used to test the application, the tester can provide multiple invalid inputs for each parameter or add new parameters. The results are presented by the percent of their effectiveness for each subject in figure 5.4 graph B (Valid vs. Invalid Responses). The types of responses are grouped by valid and invalid responses. Invalid responses include the fault, failures, and exposures.

Security Evaluation

This empirical evaluation of bypass testing was not specifically tailored to expose security flaws in the subject applications, yet, it is important to note that in several cases invalid input did pass to the back-end of applications without validation. After careful study of this behavior, malicious users could potentially exploit invalid input vulnerabilities and compromise the security of applications.

For instance, applications that use server side scripting technologies (e.g. ASP or PHP) can be vulnerable when scripts are passed to the application in the form of invalid inputs and execute on the server. One example of this behavior was the invalid inputs on the Joomla application, which passed through to the output for the interface's navigation. In other cases, (e.g. phpMyAdmin) invalid input reached the database queries with no validation. Finally, session related values, such as the `VIEWSTATE` parameter of .NET applications, which is stored in hidden fields, are often

passed without validation. In such cases, the values out of range were usually caught by the server with an exception handling mechanism, but theoretically one can access the session of another user by manipulating the value of this parameter.

Effectiveness of Violation Rules

An interesting aspect of this study arises from the comparison of the effectiveness of the different violation rules used to generate test cases. The overview of results listed by violation rules is presented in table 5.35. Three quarters of the test cases (75.3%) were generated by value violation rules. Field element violation rules produced 15% of the test cases and 8.5% were generated by target URL violation rules. The remaining 1.2% were generated by length and transfer mode violations. There is a significant difference among the number of test cases relative to the rules used to generate them. Similarly to the variation among the numbers of test cases for each subject, the amount and type of input controls found in each subject is directly proportional to the number of test cases generated by each rule. For instance, only a few of the interfaces use *maxlength* constraints; therefore, few test cases are generated by this rule.

The results with respect to each violation rule is presented in figure 5.5. It is shown that length violation rule produced invalid responses with the greatest rate (55.5%), but only 0.6% of the test cases are generated using that rule. Next, the value and the field element violations produce invalid responses with a rate of 28.8% and 12.1% respectively. Finally, the least effective rules in this experiment are the transfer mode and target URL rules.

Table 5.35: Result Summary by Violation Rules

Violation Rule	Response Type			Total
	Valid	Faults and Failures	Exposure	
Length Violation	12	10	5	27
Value Violation and Invalid Characters	2581	511	531	3623
Transfer Mode	30	1	0	31
Field Element Violation	635	53	35	723
Target URL Violation	405	3	0	408
Total	3663	578	571	4812

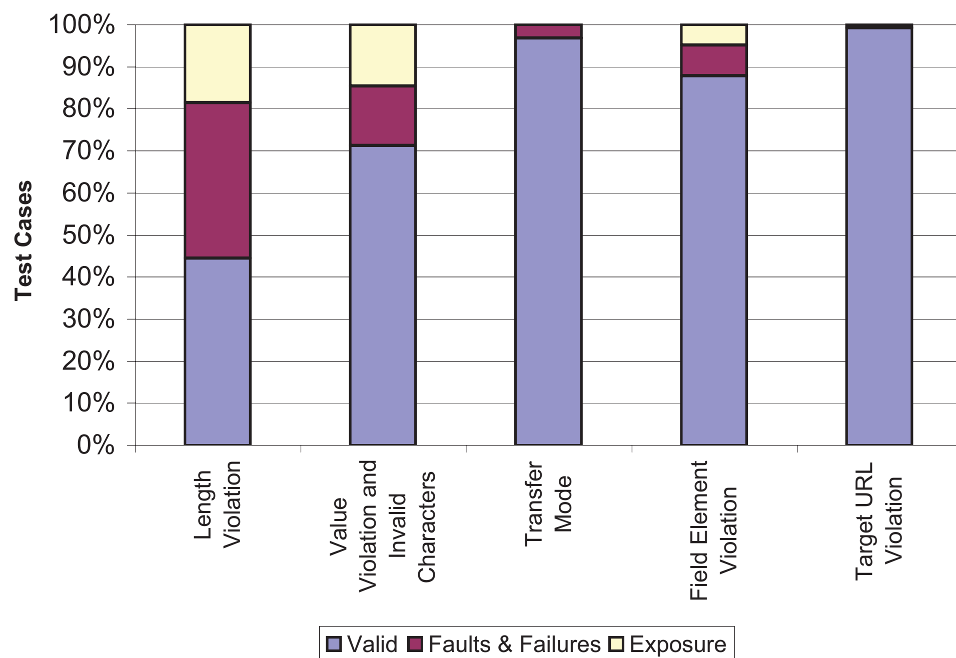


Figure 5.5: Responses by Violation Rule

Testing Cost

One important aspect of bypass testing is the amount of time and effort put in to performing the testing using the automated tool. Softaware quality has become a major aspect of software evaluation [13] [15]. In fact, developers invest their assets to perform adequate testing on their products to improve quality. Quality assurance and testing efforts can account for more than 50% of the development cost [9] [8], which can be translated to hundreds or even millions of dollars in some cases. The level of effort put during the experiment is provided in table 5.36. It is shown that bypass testing performed by an automated tool can expose faults in a fraction of time and cost of standard testing techniques. In fact, the subject applications were tested in a matter of hours and in some cases major faults were found. And this was after the applications were tested and deployed.

5.4 Confounding Variables

There are several external factors that could have influenced these results. A major component of this study is the automation tool, which introduces some confounding variables. Other variables include the subjects selected, the tester's inputs, and the results evaluation. These issues are addressed in the following subsections.

5.4.1 Effects of AutoBypass Implementation

Prior to acquiring the test results that are listed in this paper, the author has performed multiple test sessions to validate the test generation and submission by the

Table 5.36: Level of Effort

Application	Subject ID	Tool Processing Time *		Tester Time *		Total Time	
		Test Case Generation	Test Case Submission **	Creating Input Parameters	Reviewing Responses	minutes	hours
atutor.ca	S01A	1	41	5	184	**231**	**3.8**
demo.joomla.org	S02A	1	5	10	426	**442**	**7.4**
	S02B	1	2	10	43	**56**	**0.9**
phpMyAdmin	S03A	1	2	10	27	**40**	**0.7**
	S03B	1	3	15	51	**70**	**1.2**
	S03C	1	5	15	67	**88**	**1.5**
	S03D	1	7	5	63	**76**	**1.3**
brainbench.com	S04A	1	2	5	76	**84**	**1.4**
	S04B	1	8	10	281	**300**	**5.0**
myspace.com	S05A	1	3	5	77	**86**	**1.4**
	S05B	1	1	5	31	**38**	**0.6**
nytimes.com	S06A	1	5	10	176	**192**	**3.2**
mutex.gmu.edu	S07A	1	1	2	46	**50**	**0.8**
yahoo.com	S08A	1	1	10	49	**61**	**1.0**
	S08B	1	15	15	235	**266**	**4.4**
	S08C	1	1	1	19	**22**	**0.4**
	S08D	1	2	2	30	**35**	**0.6**
barnesandnoble.com	S09A	1	2	5	88	**96**	**1.6**
	S09B	1	2	5	71	**79**	**1.3**
amazon.com	S10A	1	3	5	32	**41**	**0.7**
	S10B	1	1	5	17	**24**	**0.4**
bankofamerica.com	S11A	1	8	5	64	**78**	**1.3**
	S11B	1	5	10	105	**121**	**2.0**
comcast.com	S12A	1	2	5	53	**61**	**1.0**
ecost.com	S13A	1	1	5	32	**39**	**0.6**
	S13B	1	6	10	80	**97**	**1.6**
google.com	S14A	1	1	10	74	**86**	**1.4**
	S14B	1	2	10	54	**67**	**1.1**
pageflakes.com	S15A	1	5	10	101	**117**	**1.9**
wellsfargolife.com	S16A	1	3	10	152	**166**	**2.8**
	Total	**30**	**145**	**230**	**2798**	**3203**	**53.4**
	Average	**1**	**5**	**8**	**96**	**110**	**1.8**

* Time in Minutes

** Time it took the tool to sumbit the test cases over HTTP and capture responses

tool. A sample web application was created to test the AutoBypass tool, which represents the subject application, and used to analyze all the inputs provided by the tool. It is certain that the test cases generated by the tool and reported on the output file matched the requests that were actually sent to servers. Thus, the request submissions by AutoBypass is unlikely to have affected the results.

5.4.2 Sample Web Applications Selected

Another factor that may have influenced the experiment results is the limitation on the tool to examine interfaces that use extensive Javascript or technologies other than HTML. Some subjects were excluded from this study for these reasons (e.g. `cnn.com`, `ebay.com`, and some tools by `google.com`). In some others, various methods where used to parse an interface that was creating problems with the parser, such as saving the target interface to a file and modified the HTML source to create a simple interface that can be parsed by AutoBypass. This issue has the effect of limiting the subjects that could be selected as well as adding to the cost.

The parsing mechanism of AutoBypass uses the HTML form attribute *name* to separate and identify different forms in a web pages. Some applications, e.g. ATutor, do not always use this attribute for the form elements; therefore, form parsing was done by copping the HTML file to a local disk and modifying it by giving a name to the form. In such cases, AutoBypass is parsing the local file while it still submits the invalid requests to the remote server as described by the form action attribute. Indeed, this had an effect on the complexity on the testing of certain subject. Yet, the fact that the tester in this experiment had no access on the source code or the documentation of the applications is a disadvantage for the amount of time and effort

required for testing. It is expected that testers with knowledge of the application domain and control over the source code can eliminate such obstacles and produce test cases more efficiently.

For the joomla CMS test set, a variable that controls the authentication mechanism, caused the application to log the user off (in this case AutoBypass). The remaining responses did not represent the actual test cases. These were examined by authenticating a session on a browser and then submitting the mutated forms manually. This issue caused AutoBypass to log off the subject application and the rest of the tests were submitted manually. A similar case was `brainbench.com`, in which every submission must have a different email address. Otherwise, the application responded with a message that the email is already used to create an account (as previous test cases submited the same values). These types of issues were addressed by the ability to use the stored mutants for testing.

The author tried to avoid unexpected results from sensitive commercial applications (e.g. banking) Commercial applications were tested using components that should not affect the business in case of a failure. These limitations on the subject selection may have influenced the results.

5.4.3 Test Values Input

The tester, which is the author in this study, plays a major role in performing the tests. The values selected to violated the constraints in the interfaces can greatly affect the results. For this experiment, no domain knowledge was applied for producing specific invalidated requests. Most of the values generated were complements of the default values of the controls. For instance, if a hidden control had a numeric default value,

then an arbitrary character string was generated. The author believes that testers that are familiar with the application (such as its designers and developers) can generate more effective invalid input selections, which would have possibly exposed more faults.

5.4.4 Result Evaluation

Detecting faults and failures in server responses is a challenging process. According to the framework defined in the previous chapter and specifically the experiment design, certain rules were used to evaluate the responses of subject applications. Indeed, the appropriateness of responses to invalid input can be a subjective matter. The author tried to classify results to the best of his knowledge. It might be useful to have multiple reviewers to allow for a cross rater evaluation of the results.

Result evaluation, as mentioned before, is performed by inspecting the output of the test requests. No access to the source code, storage, databases, or server logs, is available to better evaluate the effects of the invalid inputs. For example, during the development of the AutoBypass tool, a small web application that is used for student records was tested. After approximately 50% of the test cases were submitted, the Tomcat server produced an exception report, and no other response was received thereafter. The invalid input had caused a major error on the XML parser, which resulted in the deletion of all the data of the application. In addition, the tomcat service exited abnormally due to the handling of this invalid input. These facts would not be available if access to the application servers was granted.

An other level of result verification was the manual submission of mutated forms using a browser. For results that suggested that faults or exposure may have been

due to AutoBypass, a browser was used to submit the HTML version of the mutated forms that were stored for each test case. Two browsers were used for this purpose, Internet Explorer version 6 and Firefox version 1. No differences were identified by submitting mutated forms with these browsers.

Finally, since AutoBypass provides the ability to generate test cases and submit them to the server, a relatively large number of results were collected. However, counting responses that were submitted through similar or identical requests through different testing sessions would have affected the results. For that reason, the author compared the results against the violated parameters of different test sets and eliminated duplicates, which significantly reduced the number of the results presented in this paper.

Chapter 6: Conclusions

Upon completion of this experiment and collection of the data, it is found that the hypothesis was rejected. Bypass testing **can** reveal errors in web applications beyond what standard testing can find. It is clear that web applications can benefit by adopting this method as a strategy to ensure higher quality products. Bypass testing is based on limited test case generation that is very well targeted to expose application faults. Therefore, with only limited resources, organizations can integrate bypass testing with the standard testing strategies to accomplish higher test coverage for applications.

The sample used in this study provides a small representation of professionally developed web applications from which the author tries to draw some conclusions. Yet, the goal of this study is not to prove that web applications are not well built. Instead, the goal is to develop an efficient method of revealing vulnerabilities that can be potentially used by malicious users to exploit web systems. Numerous cases of incomplete input validation were found; due to the constraints set in the interface, the values are taken for granted by the developers.

Not all of the subjects behave the same way and therefore the author cannot generalize these results for all web applications. From the results it is shown that advanced applications with millions of users such as `google.com` and `amazon.com` revealed no errors. On the other hand, application with fewer users revealed a significant number of errors (e.g. `brainbench.com`, `mutex.gmu.edu`, `ecost.com`, and

`wellsfargolife.com`). This variety demonstrates that development firms that devote substantial resources in testing their application are not affected by the test cases created using bypass and vice versa.

The automated approach of AutoBypass efficiently creates tests cases that recovered the invalid responses by the subjects. On the other hand, it is hard to automatically evaluate the application responses. Since that the applications' behavior for invalid inputs is not specified, the response evaluation can not be performed programmatically (e.g. by comparison of the output to the invalid input to a known valid input). Instead, a human is required to evaluate the appropriateness of the output given the invalid inputs.

Indeed, bypass testing can greatly improve the quality of web applications. Even in a final product, potential vulnerabilities can be identified. Moreover, the author strongly believes that bypass testing and the automated approach can be more effective when used early in the development of web applications by the developers, in a controlled environment with access to the source code, knowledge of the application domain, and system resources.

6.1 Future Work

In order to develop the AutoBypass tool further, improvements on the tool must address the implementation of scripting rules, overcome the problems with parsing interfaces, and the problems resolving complex relative URLs. Next, a mechanism to allow testing on a sequence of events shall be developed. A major characteristic of AutoBypass is performing tests in a single form at its initial state, without the ability to test a sequence of events. Such approach can include the parsing of entire

web sites to identify possible paths for testing. That would also benefit the creation of a large input domain form parameters found throughout an application.

Finally, the optimization of the number of test cases generated can be evaluated. Given a set of valid inputs for a web applications, AutoBypass generated test cases must be filtered to exclude valid ones, which will produce valid responses. Extra parameters may be added for display, which will not affect the processing. For instance, "`http://google.com?`", "`http://www.google.com/ig?hl=en`", and "`http://www.google.com/ig?`" are all in the valid input domain of the application. In future, when complete sites are parsed and valid input domains are generated, AutoBypass should avoid creating such requests.

References

References

[1] *AJAX, The official Asynchronous Javascript And XML special interest group site*. Online: http://ajax.org/, last access May 2006.

[2] *API Specification, Java(TM) 2 Platform, Standard Edition, v 1.4.2.* Online: http://java.sun.com/j2se/1.4.2/docs/api/, last access April 2006.

[3] *ATutor, an Open Source Web-based Learning Content Management System (LCMS).* Online: http://www.atutor.ca/, last access May 2006.

[4] *The Joomla Project home.* Online: http://www.joomla.org/, last access December 2005.

[5] *The phpMyAdmin Project home.* Online: http://www.phpmyadmin.net, last access March 2006.

[6] Amalio Saiz de Bustamante and Aniello Amendola. *Reliability Engineering.* Springer, July 1988.

[7] Russell Gold. *HttpUnit home.* HttpUnit, Online: http://httpunit.org/, last access November 2005.

[8] Mary Jean Harrold. Testing: A roadmap. In *International Conference on Software Engineering*, pages 61–72, Limerick, Ireland, June 2000.

[9] Thomas Hilburn and Massood Towhidnejad. Software quality across the curriculum. Proceedings of the 15th Conference on Software Engineering Education and Training, February 2002.

[10] James L. Johnson. *Probability and Statistics for Computer Science.* Wiley-IEEE, July 2003.

[11] E. Nebel and L. Masinter. Form-based file upload in HTML. Request For Comments 1867, Network Working Group, Online: http://www.ietf.org/rfc/rfc1867.txt, last access April 2006., November 1995.

[12] Rae R Newton and Kjell Erik Rudestam. *Your Statistical Consultant.* Sage Publications Inc, January 1999.

[13] Jeff Offutt. Quality attributes of web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, 2002.

[14] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 187–197, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Leon Osterweil, Lori Clarke, Richard DeMillo, Stuart Feldman, Bill McKeeman, Edward F. Miller, and John Salasin. Strategic directions in software quality. *ACM Comput. Surv.*, 28(4):738–750, 1996.

[16] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification - W3C Recommendation 24*. World Wide Web Consortium (W3C), Online: http://www.w3.org/TR/html401/, last access February 2006, December 1999.

[17] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software*. World Wide Web Consortium (W3C), Online: http://www.dwheeler.com/secure-programs/, last accessed November 2005, 3 edition, March 2003.

[18] James D. Wynne. *Learning Statistics, A Common-Sence Approach*. MacMillan Publishing Co., Inc., New York, 1982.

[19] Wei Xu, Sandeep Bhatkar, and R. Sekar. A unified approach for preventing attacks exploiting a range of software vulnerabilities. Technical Report SECLAB-05-05, Department of Computer Science, Stony Brook University, Online: http://seclab.cs.sunysb.edu/seclab1/pubs/papers/seclab-05-05.pdf, last access February 2006, August 2005.

Curriculum Vitae

Mr. Papadimitriou has over six years' experience as an information technology manager and in developing software systems for web-based applications. He specializes in GUI design and development, and his research interests include testing, usability, web software engineering, and service-oriented architectures. He offers expertise in advanced planning and coordination of complex projects, managing project financials, assuring quality systems, and managing teams. His computers skills cover a variety of platforms and technologies and a range of software tools for software development, and graphics.

Mr. Papadimitriou is currently working towards a M.S. Degree in Software Engineering. He has earned a B.S. in Computer Science from George Mason University in 2004 and a certificate in applied arts and design from Vakalo School of Art & Design (Athens, Greece) in 1996. His professional experience include multiple positions as a project manager and software developer, are well as architectural and graphics designer. Mr. Paparimitriou is a member of the IEEE Computer Society.