Master Thesis Software Engineering

Intelligent fuzzing of web applications

Michel de Graaf
michel@re-invention.nl

Universiteit van Amsterdam

Supervisor:
Tijs van der Storm

Publication status:
Public domain

September 15, 2009

Abstract

Fuzz testing (also known as fuzzing) is a blackbox testing technique for finding flaws in software by feeding random input into applications and monitoring for crashes.

Programs that generate fuzz data are called fuzzers and they generate input data that test engineers might not think of. There are two categories of fuzzers, unintelligent (UF) and intelligent (IF). The difference lies in the method of input data generation. UF has no prior knowledge of the input format while IF knows the format which enables it to specify semi-valid data for what its attempting to fuzz.

Sources like [21, 20] have indicated that user input in web applications are a huge problem. Fuzzing might prove to be a valuable method for finding flaws in these types of applications. However, the research that has been done on fuzzing web applications [6] have made use of UF. In this thesis we will introduce and evaluate an IF method based on validators.

Many modern web applications are developed using specialized web frameworks that make use of validators that validate incoming input before further actions are taken by the application.

Our hypothesis is that the data generated by a UF will often be evaluated as invalid by validators that are in place and will therefore have superficial code coverage. Intelligent fuzz data that is generated within validator constraints will have better code coverage and will therefore trigger more flaws.

In order evaluate the effectiveness of our IF method we have fuzzed a set of typical web applications using 3 different fuzzing methods: UF, our IF method and fuzzing with manually defined fuzz format specifications.

The results of this experiment indicate that our method of intelligent fuzzing performs marginally better while requiring more manual effort. This manual effort can be further automated, which would make it a valuable addition to fuzzing web applications.

Keywords: Fuzzing, Intelligent fuzzing, web applications

Contents

List of Figures

List of Tables

Listings

Preface

I would like to thank my supervisor: Tijs van der Storm, for his feedback and advice during the writing of this thesis. I would also like to thank Harm de Laat and Ralph Deguelle of Kabisa ICT for allowing me to work at their office and letting me use their web applications for experiments. Of course I have to thank my parents for supporting me this last year (and the other 24 years for that matter).

I am also very grateful to Mark Sanger, my internet friend from across the pond, for finding lots of linguistic mistakes in this thesis.

And at last, but not least, i'd like to thank my classmates Lars de Ridder, Jeroen van Schagen and Alex Hartog for making the time more enjoyable. Together we have solved problems with LaTeX, gave each other feedback, and of course, performed some high quality procrastination in the form of useless discussions and sharing the loot of our visits to the wild wild internets.

Chapter 1

Introduction

Fuzz testing (also known as fuzzing) is a popular blackbox testing technique for feeding random input into applications. The criteria for reliability are simple: If the program hangs or crashes it fails the test, otherwise it passes. [15]

Programs that generate fuzz data are called fuzzers and they generate input data that test engineers might not think of. Test engineers often make implicit assumptions about the data that will or can be fed to the application under test. Fuzzers will try anything which makes fuzzing a crude but cost effective technique for finding flaws in software.

1.1 Motivation

Fuzzing has successfully been used to discover flaws in a wide set of software applications [15, 16, 14, 4]. This is evidence that many applications are not hardened enough to handle random data.

With the current growth of demand in web applications and market pressure that demands very short time-to-market, the testing of web applications is often neglected by developers [12]. And in its current form is considered too time consuming and lacking a significant payoff [7]. This triggers the need for efficient and cost effective testing methods.

Several sources like [21,20] have indicated that user input in web applications are a huge problem. Therefore, fuzzing might be a valuable addition for the

detection of flaws in web applications.

When an application doesn't do what it is supposed to do and this unwanted behavior is observable, a failure has occurred. A failure is caused by a defect in the logic of the application. In this thesis we will use the term flaw and defect interchangeably.

Flaws in web applications are triggered by user input that puts the web application in an undefined state causing unwanted behavior. For example: A user fills in a web form that contains exotic characters that will trigger an uncatched exception after processing. As a result all the text the user typed in is lost and the user has no clue what he did wrong.

There has been limited research on the effects of fuzzing web applications. The research that has been done by Hammerslands [6] shows that web applications are indeed vulnerable to fuzzing. However, the fuzzer used in Hammerslands research is of the most basic form, that is: randomly generate data and submit it to an application with no prior knowledge of the format. These are called unintelligent fuzzers (UF). Intelligent fuzzers (IF) know the format, which enables it to specify semi-valid data for what its attempting to fuzz.

This brings up an interesting question: what form of fuzzing will be more effective in detecting flaws in typical web applications?

We could see that an IF could potentially proceed further along code paths and get better code coverage and thus find more flaws. However, UF make no assumptions about the input data and from this perspective it's convincing to think such an approach could cover far outside the bounds of what the developers of the application anticipated [9].

Many modern web applications are developed using specialized web frameworks with the premise of increased maintainability and productivity. Many of these frameworks make use of validators that validate incoming input before further actions are taken by the application. One could see that when fuzzing a web application that has validators in place, most of the fuzzing attempts will be stopped by the validators as they are intended to stop invalid input from entering the application flow.

Our hypotheses is that when typical web applications are fuzzed with fuzz-data that is generated within the constraints of the validators that are in place, more and different kind of flaws will be found than pure random fuzzing.

Hypothetically speaking, when you use an UF fuzzer and let it fuzz an application for an unlimited amount of time, eventually the same fuzz data that will be generated by a IF will be generated by a UF since the possibilities of random data are infinite. Or to quote the infinite monkey theorem[1]:

> If you put an infinite number of monkeys at typewriters, eventually one will bash out the script for Hamlet.

For fuzzing to be cost effective we want to detect as many flaws as is possible in a limited amount of time using a limited amount of resources.

1.2 Research Questions

In this thesis we will introduce and evaluate a method of fuzzing web applications that is based on intelligent fuzz data. This fuzz data is generated within pre-condition constraints that are extracted from the application under test. With this method we are looking to answer the following questions to evaluate the effectiveness of IF using validators as fuzz data specification.

- Q1: Will more flaws be found in typical web applications using IF than UF.
- Q2: Does IF find different flaws than UF?

The rest of this thesis is structured as follows: First we will discuss the background and successes of fuzzing and related research. After that we will introduce the notion of intelligent and unintelligent fuzzers and how validators can be used to generate semi-valid fuzz data. Then we will discuss our research method and introduce our IF implementation called YAFT. This will be followed by the results of our experiments and finalized with an analysis and some interesting ideas for future work.

[1]The infinite monkey theorem: `http://en.wikipedia.org/wiki/Infinite_monkey_theorem`

Chapter 2

Background

The idea of fuzzing is originated by professor Barton Miller at UW Madison in 1989. It came up while he was logged in to a remote terminal that had connection issues due to noise on the telephone line. The line noise mangled up terminal commands and caused common UNIX programs to crash. This insight triggered the first fuzzing research as an assignment for his graduate Advanced Operating System class in 1990.

His students managed to crash one third of common command-line utilities by feeding them random input in seven different versions of UNIX [15].The research was repeated in 1995 [16] with the addition of X-Window GUI applications. The results showed improvement in the reliability of basic UNIX utilities but there were still significant rates of failure (9%). From the 38 X GUI applications that were fuzzed, 58% crashed, indicating that GUI applications are more error prone. Miller et al. has since then done similar research on the reliability of Windows NT [4] and Mac OS X [14] and they showed the same kind of results.

Fuzzing has become particularly popular in the software security industry for finding vulnerabilities. The flaws that are found when fuzzing applications that are written in unmanaged languages (e.g. c and c++) are often effecting memory safety and result in buffer overflows that are likely to lead to severe vulnerabilities. The detection of these vulnerabilities is the main motivator of fuzzer development and research.

2.1 Practical fuzzing successes

Big software vendors such as Microsoft [11, 8] and open source projects (e.g. GNU/Linux [13]) have integrated fuzzing into their quality assurance processes. Also the research on fuzz-testing that is done by initiatives such as "Month of BrowserBug (MOBB)"[1], "Month of the Kernel Bugs(MOKB)"[2],"Month of PHP Bugs"[3] and "Month of Apple Bugs"[4] have especially shown the effectiveness of fuzzing software by releasing new security vulnerabilities each day during a month.

Beside conventional software applications, fuzzing has been successfully applied to wireless network drivers [10] and networking protocols [1]. The most recent high profile exploit that has been discovered using fuzzing techniques, and made public at BlackHat 2009[5], is the iPhone SMS exploit[6]. This exploited flaw allowed attackers to execute arbitrary commands as a root user effectively taking ownership of the device.

2.2 Fuzzer types

As mentioned in the introduction, there are so called unintelligent (UF) and intelligent fuzzers (IF). The difference is that UF have no prior knowledge about the data format and IF do have knowledge about the data format. Since IF have knowledge about the format they can produce semi-valid/invalid data (semi-valid is the same as semi-invalid in this context). When this semi-valid data is fed to an application it should potentially proceed further along code paths and get better code coverage and therefore trigger more flaws.

As Howard et al. mentions [9], there is always a trade-off between IF and UF. IF takes more work to implement and makes assumptions about the format. This potentially weakens the fuzzers efficiency to detect flaws since it could make the same poor assumptions the programmers did.

[1]Month of Browser Bugs: `http://browserfun.blogspot.com/` (2006)
[2]Month of Kernel Bugs: `http://kernelfun.blogspot.com/` (2006)
[3]Month of PHP Bugs: `http://www.php-security.org/` (2007)
[4]Month of Apple Bugs: `http://applefun.blogspot.com/` (2007)
[5]BlackHat security conference: `http://www.blackhat.com/`
[6]Blackhat 2009 iPhone SMS exploit presentation: `http://www.blackhat.com/presentations/bh-usa-09/LACKEY/BHUSA09-Lackey-AttackingSMS-SLIDES.pdf`

2.3 Semi-valid fuzz data generation

UF generates pure random data and IF generates semi-valid data. There are two main methods for generating semi-valid data. The first method is called *mutation-based* fuzzing. Mutation based fuzzing takes known good collected data (files, captured input, network traffic, etc) and then modifies it. These modifications (or mutations) may be random or heuristic. Examples of heuristic mutations include replacing small strings with longer strings or changing integer values to either very large or very small values [17].

The other method is called *generation-based* fuzzing. Generation-based fuzz data starts from a specification/format description and constructs fuzz data from these specifications. The main point is to generate effective and diverse fuzz data that (hopefully) will invoke a flaw in the application under test, but not make the data invalid to the degree that it causes the application under test to discard the incoming data [17].

2.4 Fuzz data for web applications

There is a wide range of tools that support fuzzing web applications but there are no empirical studies of their findings. Some of these tools (e.g. Burp Intruder[7] and WSFuzzer[8]) claim to use fuzzing techniques for detecting SQL injection[9] (SQLi), cross site scripting[10] (XSS) and command injection[11] vulnerabilities. However, these tools are enumerating known vulnerable strings. Enumeration testing might be a better way to find these kinds of flaws but they are not to be confused with fuzzing.

In this thesis we will make use of generation-based fuzz data to fuzz web applications. We have chosen this method over mutation-based fuzz data because it requires less manual effort to generate fuzz data than to collect and mutate known valid data.

In order to mutate data to semi-valid data you will need valid data. But how can we collect valid data in a web application? Let's presume that

[7] Burb Intruder: `http://portswigger.net/intruder/`

[8] WSFuzzer: `http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project`

[9] SQL injection: `http://www.owasp.org/index.php/SQL_Injection`

[10] Cross site scripting: `http://www.owasp.org/index.php/Cross_Site_Scripting_Flaw`

[11] Command injection: `http://www.owasp.org/index.php/Command_Injection`

we will use web forms as an entry point for valid data in web applications. Then there is still a need for an oracle that can evaluate the validness of the entered data.

Fuzzers like OWASPs[12] WebScarab[13], Suru[14] and SPIKE Proxy[15] solve the problem of harvesting valid input data by introducing a man in the middle proxy server that captures HTTP[16] requests that are sent from a web browser operated by a human tester that submits valid data.

Our IF will generate semi-valid data by using validators that are extracted from the target application as a specification for the generation of semi-valid fuzz data. This process can be automated so no interaction from a human tester is required.

[12]The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Online at: `http://www.owasp.org`

[13]WebScarab: `http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`

[14]Suru: `http://www.sensepost.com/research/suru/`

[15]SPIKE Proxy: `http://www.immunitysec.com/resources-freesoftware.shtml`

[16]HTTP - Hypertext Transfer Protocol: `http://www.w3.org/Protocols/`

Chapter 3

Research method

In the previous chapter we introduced the notion of IF. Now is a good time to step back and see how we can apply this to web applications.

First we will discuss the fuzz entry points of web applications. Then we will describe how we will generate intelligent fuzz data using validators as fuzz data specification. Finally, we will discuss how we will analyze the fuzzing results and categorize detected flaws.

3.1 Intelligent fuzz data generation using validators

As indicated by [21, 20], user input is a problem in web applications. The primary way for users to insert data into web applications are forms. Forms are also, the place where flaws tend to propagate after execution. Therefore, we have identified forms as the primary attack vector in this research. Other potential web application fuzzing vectors are cookies, URLs and webservices but they are left out of the scope in this research.

Forms are described in HTML and when they are submitted by the user they are sent by the browser over the HTTP protocol to the web application. When input data arrives at a web application it is typical that the web application takes some steps to check if the incoming input data is valid before other actions are taken (validation input). When the incoming data is found to be invalid, the application will return some warning message and

discard the invalid data. The functions that evaluate if the incoming data is valid are called *validators* in this thesis.

Forms typically contain one or more fields and validators are used to validate individual fields. If one of the fields is evaluated as invalid the whole submitted form is seen as incorrect and the application will not proceed further along the projected code path.

The fuzzer used in the research of Hammersland [6] is primarily aimed at forms. But the input data that it generates is pure random. Therefore, we can categorize this fuzzer as unintelligent. Our hypothesis is that the data generated by Hammersland's fuzzer will often be evaluated as invalid by validators that are in place and will therefore have superficial code coverage.

Many modern web applications are developed using some kind of web specific framework. Many of these frameworks have dedicated validator functions that can be used to validate common input formats. We propose that these validators can be used to create an intelligent form fuzzer that has a better code coverage and will therefore potentially trigger more flaws.

In order to answer research question Q1 we have chosen to test a set of typical web applications written in the popular open source web-framework Ruby on Rails[1]. Ruby on Rails is a state of the art web-framework written in the dynamic programming language Ruby[2]. The framework has a strong set of conventions that simplify the identification and extraction of validators that belong to a specific form field.

Ruby on Rails has a Model View Controller (MVC) architecture that divides application logic (Model) from output (View) that are tied together by controllers. The models in Ruby on Rails are mapped to database tables using Object-relational mapping (ORM) [18]. In the models, the validators are defined using a Domain-specific language (DSL) [2] embedded in Ruby.

A DSL is a specialized language for a particular domain. An embedded (or internal) DSL is a DSL that is implemented using a subset of syntax of a general purpose language such as Ruby.

Listing 3.1 gives an example of arbitrary validators that might be defined on a user model. In this example you see a user model that has 3 validators. Validators are mapped to form fields and database columns with the same name.

[1]Ruby on Rails: `http://rubyonrails.org`
[2]The Ruby programming language: http://ruby-lang.org

```
1  class User < ActiveRecord::Base
2    validates_confirmation_of :password
3    validates_length_of :username, :within => 3..40
4    validates_format_of :email,
5    :with => /\w@\w\.\w{2}/
6  end
```

Listing 3.1: Arbitrary validators in a Ruby on Rails model

Line 1 of listing 3.1 shows the sub-classing of `ActiveRecord::Base` (a Ruby on Rails library for ORM mapping) to create an ORM mapping to the database table `users`.

On line 2 the validator `validates_confirmation_of` is defined that requires that the value of an incoming form field `password_confirmation` is identical to the value of the form field `password`.

Line 3 shows a validator that validates that the form field `username` has a length within 3 to 40 characters. And finally, in line 4+5 we have an example of a validator that uses regular expressions[3] (in this case `/\w@\w\.\w{2}/`) to validate the format of the form field email.

These validations are triggered when data is requested to be persisted in the database. This is typically the case when incoming form requests are handled. If one validator validates incoming data for a field as invalid than the data will not be persisted in the database and the end-user is presented with a warning message. This warning is not to be seen as a flaw since the invalid input is correctly handled and the user is presented with a graceful warning message.

3.2 Analysing web fuzzing results

As indicated by Hammersland [6] and Stuttard et al. [19] analyzing the results of web application fuzzing is a hard problem to automate. When fuzzing non-web applications it is possible to attach debuggers and monitor the target application for unexpected behavior (crashes, hangs). This is not the case in web applications where incoming requests are typically served by a web application server that does not crash when a fault is triggered in the web applications logic. Instead, they return HTTP status codes as

[3]Regular expressions: http://www.regular-expressions.info/

described in section 10.4 of RFC2616[4].

The status codes in the 5xx range are reserved for requests where "the server failed to fulfill an apparently valid request"[5]. Well-behaved web applications will return the appropriate status code when a triggered defect escalates beyond rescue.

The fuzzer used in this thesis will monitor the returned HTTP status codes and will report responses with the status code in the 5xx range as erroneous. In combination with the error logs from the web server that contain stack traces of crashes, we can determine the cause of the defect.

3.3 Expected flaws and categorization

We categorize detected and identified flaws in the same categories as Hammerslands fuzzing research [5]:

- **E1** *Resource exhaustion:* This category manifests when flaws cause increased response time and possibly no response at all. This is often caused by endless loops or non-terminating recursion.
- **E2** *Failure to check return values:* Caused by not catching exceptions. Causing the user to see cryptic exception messages and stack traces.
- **E3** *No server side validation of input:* Caused by not validating or sanitizing incoming data. Depending on the flaw the user will experience this the same way as category *E2.*

In this research we can only detect flaws that will immediately propagate after a fuzzing attempt. However, faults can propagate in other locations of the target web application. A primary example of these kind of flaws are flaws that are caused by insufficient data validation. Data of a fuzzing attempt is persisted in a database and later used on another page that contains a function that attempts to parse the persisted fuzz data but fails, resulting in an uncatched exception.

A possible solution for this problem is to crawl the entire web application and store the URLs of unique pages before a fuzzing attempt. And then,

[4]RFC2616: `http://www.ietf.org/rfc/rfc2616.txt`
[5]RFC2616: `http://www.ietf.org/rfc/rfc2616.txt`

after the fuzzing attempt, visit the earlier stored URLs to see if any of them return an erroneous response. We leave this implementation as future work.

Chapter 4

YAFT: Yet Another Fuzzing Tool

In this chapter we introduce the fuzzer we developed for this thesis. It is written in the dynamic programming language Ruby and makes use of several open source (Ruby) libraries (also, known as gems in the Ruby world). The fuzzer could just as well be implemented in other languages and make use of existing fuzz frameworks such as Sully[1] ,Spike[2] or Peach[3]. However, since we are concentrating our experiment on web applications written in Ruby on Rails, we prefer the flexibility and support Ruby provides us when interacting with Ruby on Rails applications.

YAFT (Yet Another Fuzzing Tool) has support for unintelligent (random) and intelligent fuzz data generation. It generates test cases (we will call them attack scripts) that are described in an embedded ruby DSL allowing the tester to manually define or fine-tune the fuzz data specifications of test cases.

YAFT is divided in 3 standalone command line applications as seen in figure 4.1. When used together they semi-automate the process of fuzzing. In the upcoming sections we will discuss the individual parts of YAFT.

[1]Sully fuzzer framework: `http://code.google.com/p/sulley/`
[2]Spike fuzzer toolkit: /urlhttp://www.immunityinc.com/resources-freesoftware.shtml
[3]Peach fuzzer: `http://peachfuzzer.com/`

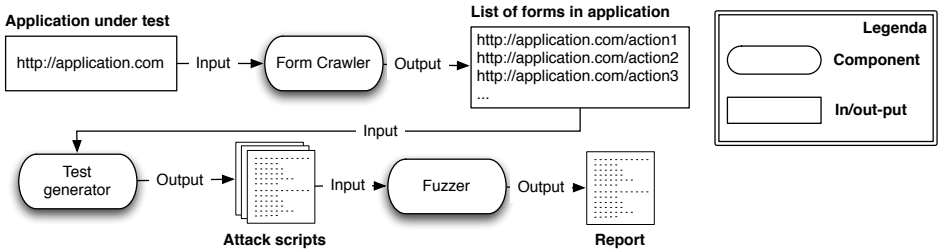

Figure 4.1: An overview of the YAFT components with their in and output.

4.1 Form crawler

The *Form crawler* component (see figure 4.1) is used to search for forms to fuzz in web applications. It simply visits every link it encounters on a page recursively and keeps track of all the unique forms it finds. It can be configured to execute actions (such as logging in) before crawling and can accept a list of excluded sub-strings that will be ignored in links when crawling. This exclude list can be used to stop the crawler from logging out when crawling as an authenticated user and triggering destructive functions such as deleting vital database records. The output of this component is a list of unique URLs where forms are located that can be fuzzed.

4.2 Script generator

The *Test generator* component (see figure 4.1) is used to generate attack scripts for web forms. It takes a list of URLs (provided by the *Form crawler*) and generates an attack script for each form it finds on the given URLs.

An attack script describes how a form will be fuzzed and is defined in an embedded Ruby DSL. Listing 4.1 shows an example of an attack script generated by the test generator.

```
1  TestForm ["http://80.255.251.93/users,"//form[@id='edit_user']"] do
2    name "new_user"
3    action "http://80.255.251.93/users"
4    method "post"
5    post_conditions :no_invalid_status_codes
6    cookie "52edd1182f32449b944b5acce65ca9a2a9b7a46f\n"
7
8    password = /.{4,40}/.gen
9
10   fields do
11       input 'return_action', random
12       input 'user[terms_of_service]', random
13       input 'account_type', random
14       input 'user[login]', /.{3,40}/
15       input 'user[email]', /\w{1,10}@\w{1,10}\.\w{2}/
16       input 'user[password]', password
17       input 'user[password_confirmation]',password
18       input 'user[first_name]', random
19       input 'user[middle_name]', random
20       input 'user[last_name]', random
21       input 'user[company_name]', random
22       input 'device_permission[technical]', random
23       input 'device_permission[channels]', random
24       input 'device_permission[history]', random
25       input 'account_reseller_id', random
26     end
27 end
```

Listing 4.1: A generated attack script

An attack script describes a form and its fuzzing specifications. A form has its own unique identifier that is formed by combining the URL location of the form and the XPATH[4] location in the HTML document as seen on line 1 of listing 4.1.

Most forms have a name defined in its HTML specification, this name is displayed in the attack script (see line 2 of 4.1). It has no other function than providing additional identification information for the tester.

Form requests can be sent in several ways over the HTTP protocol. The main methods are `GET`,`PUT`,`POST` and `DELETE`. This collection of HTTP "verbs" is heavily used in modern web applications that are build around the `REST` principles [3]. The HTTP method of sending the fuzz request is described on line 4 of listing 4.1.

[4]XPATH specifications: `http://www.w3.org/TR/xpath`

When a form has the `GET` verb as method value, the form field names and fuzzing values are encoded as part of the fuzzing URL as described in RFC1738[5] and RFC 3986[6].

Each attack script requires a defined set of post-conditions (line 5 of listing 4.1). After a fuzzing attempt they evaluate the returning HTTP response validness. By default the pre-condition `:no_invalid_status_codes` is used. This method evaluates HTTP status codes in the `5xx` range as erroneous. This alone is often not enough to identify a flaw after a erroneously flagged fuzzing attempt. The log files of the web application server often yield more interesting information in the form of stack traces and error messages.

Many forms are only available in web applications after a user has signed in. When a user is logged in on a web application, the user authentication information, and other temporal information is stored in `cookies`[7]. Cookies are used to maintain state between HTTP requests since HTTP is a stateless protocol. Meaning that each HTTP request made is fundamentally detached from requests that came before, and unrelated to requests that will follow. In order to maintain session state during fuzzing, the session cookie is stored in the attack script as seen on line 6 of listing 4.1.

During our initial testing of YAFT we encountered issues with fuzzing forms that made use of nonces. A nonce is a randomly generated, cryptographic token that is used in forms to thwart cross site scripting attacks. When a form contains a nonce and is submitted, the web application will check if the nonce is present in the submitted form before continuing any further.

This means that our fuzz request must contain a valid nonce in order to get any code coverage. We solved this problem by scanning the field names of the forms for known fields that must contain nonce tokens. When such a field is found, a special keyword is used in the attack script as the field value.

For example: In the popular blogging platform Wordpress[8] the field `_wpnonce` is used in forms across the application. When YAFTs script generator encounters such a field, it will use `input'_wpnonce', nonce("_wpnonce")` as fuzz data format description for that field.

[5]RFC 1738: `http://www.ietf.org/rfc/rfc1738.txt`

[6]RFC 3986: `http://tools.ietf.org/html/rfc3986`

[7]Cookies RFC: `http://www.ietf.org/rfc/rfc2109`

[8]WordPress is a state-of-the-art publishing platform with a focus on aesthetics, web standards, and usability: `http://wordpress.org/`

When the keyword `nonce("_wpnonce")` is executed by YAFT's form fuzzer it will first fetch the page where the form is located ,extract the nonce token and use it in the fuzz request resulting in a semi-valid form request.

4.2.1 Fuzz data generation

The main fuction of attack scripts is to describe the fuzzing format of form fields. Using this format description; semi-valid random data is generated.

The random data our fuzzer uses is generated using the RFuzz[9] Ruby library, this library uses the ARCFOUR[10] (RC4) stream cipher to generate pseudorandom data. This cipher will yield the same sequence of output when given the same seed. This makes replaying the attack scripts possible.

In order to generate semi-valid data for IF we use a subset of the regular expression language to describe the format of intelligent fuzz data. The format of our intelligent fuzz data is extracted from the validators that are present in the (ORM) models of target applications. For instance: in listing 4.2 we have a Ruby on Rails model for the class `User`. This model contains only one validator on line 2 that validates the format of the field `email` by using the regular expression `/\w{1,10}@\w{1,10}\.\w{1,2}/`.

```
1  class User < ActiveRecord::Base
2    validates_format_of :email, with => /\w{1,10}@\w{1,10}\.\w{1,2}/
3  end
```

Listing 4.2: A validator that uses a regular expression in a Ruby on Rails model

When an attack script is generated that contains the field `user[email]` our attack script generator can deduce to what model the field belongs by inspecting the source of the target application. In this case the field `user[email]` belongs to the model `user` and the validator on line 2 of listing 4.2 .

The attack script generator uses the regular expression from the validator to specify semi-valid fuzz data for the email field in the attack script. For example, the regular expression `/\w{1,10}@\w{1,10}\.\w{1,2}/` will generate the following random email address: `taurodont@overrake.eu`.

[9] RFuzz website: `http://rfuzz.rubyforge.org/`

[10] ARCFOUR/RC4/ARC4 description: `http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt`

This form of generation-based fuzz data gives us great flexibility when defining the format of individual form fields. When we want to generate pure random data with random lengths the keyword `random` is used.

More examples of fuzz data specification using regular expressions can be found in listing 4.1. This attack script is generated using the validators that are defined in the Rails model in listing 3.1. On line 14 the field `user[login]` will generate random strings with a length that is between 3 and 40. On line 16 and 17 we find the variable `password` as fuzz format specification. This variable is used there because the validator `validates_confirmation_of :password"` (see line 2 of listing 3.1) is defined on the field `user[password]`. This validator requires that the field `user[password_confirmation]` has the same value as the field `user[password]`. Therefore, a random string with a length between 3 and 40 characters is generated and stored in the variable `password` on line 8 that can be used for both fields.

4.3 Form fuzzer

The form fuzzer component is a console application that reads in attack scripts and executes them. See figure 4.2 for an overview of the in and out-put of a fuzzing attempt.

A fuzzing attempt (or fuzzing attack) can be described as follows: the YAFT fuzzer component reads in the attack script and generates fuzz data based on the specified fuzz data specifications in the attack script. It sends a HTTP request to the target application with the fuzz data and waits for a response. When a response is returned by the target application it is

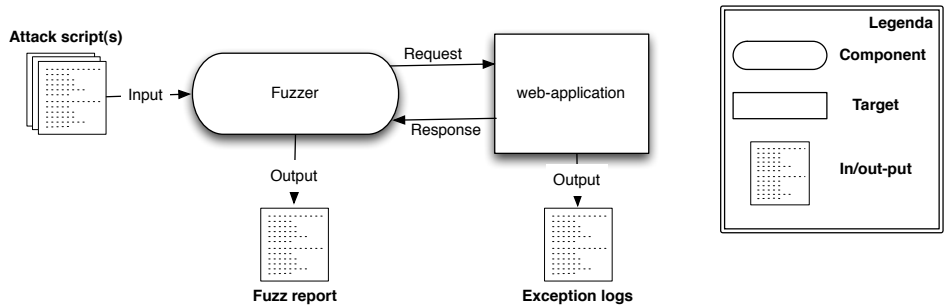

Figure 4.2: An overiew of the in- and out-put of the YAFT fuzzer component.

evaluated for validity. The validity is based on the post-conditions that are defined in the attack script.

The fuzzer will return a report with the erroneous request that were invoked during the fuzzing attempt including the fuzz data used for the fuzzing requests. This report can be used in combination with the exception logs from the target application to identify unique flaws.

In order to exercise the forms with a wide range of random fuzz data, the attack scripts must be executed multiple times. During our initial tests we tried to find an optimal amount of fuzzing attempts per test case. We noticed that when an attack script triggered flaws, they usually did not yield any other flaws no matter how many fuzzing attempts were executed (up to 10.000 attempts). Some occurrences did yield different kind of flaws but they were all detected within 100 fuzzing attempts.

For the experiment we settled on 100 fuzzing attempts per test case but further research can be done on this subject. We leave this as future work.

4.4 YAFT in action

Now that we reviewed YAFTs components we can show by example how the whole IF process works. In this short walkthrough we will start with finding forms to fuzz and end with categorizing a triggered flaw.

The target application in this example is Mephisto. This web application is also, tested in the main experiment of this thesis. See table 5.1 for more information about Mephisto.

4.4.1 Finding forms to fuzz

First we need to detect forms in the target application to fuzz. We do this by running the YAFT form crawler. See listing 4.3 for the commands and results. The flag `-pre../login_admin.rb` is used to indicate that the crawler needs to login in before crawling. This pre-condition script can be found in appendix A.1.

The flag `-exceptions "logout,delete"` is used to indicate that the crawler must not visit links that contain the sub-strings “logout” and “delete”.

```
1 $ yaft crawl http://localhost:3000 --pre ../login_admin.rb --exceptions
2 "logout,delete"
3 2 forms found on the following URLs:
4 http://localhost:3000/, http://localhost:3000/2009/7/16/hello-world
```

Listing 4.3: Searching for forms to fuzz using YAFTs form crawler

In the example on line 4 we see that the crawler has found two forms. We will use the URL `http://localhost:3000/2009/7/16/hello-world` to continue this walkthrough.

4.4.2 Generating an attack script

The next step is to generate an attack script. We do this by feeding the URL we found to the script generator as shown in listing 4.4. The flag `-rails ~/Sites/mephisto/` is used to indicate that we want to generate a intelligent attack script. It points the fuzzer to the source code of the target application so it can extract validators and translate them to fuzz format specifications.

The generated attack script can be found in appendix B.1.

```
1 $ yaft generate http://localhost:3000/2009/7/16/hello-world --rails ~/Sites/mephisto/
2 One form found
3 Generated test case in ~/forms_localhost30002009716hello-world_14a5d39.rb
```

Listing 4.4: Generating an attack script

4.4.3 Exercising forms

Now that the attack script is generated we can start exercising the form that is described in the attack script. In this example we will run the attack script only once by using the flag `--times 1`. By default an attack script will be executed 100 times in order to expose the form to a wide range of fuzz data. See listing 4.5 for the console output of the fuzzing attempt.

```
1 $ yaft exercise forms_localhost30002009716hello-world_14a5d39.rb
  --times 1
2  INFO: Found 1 form(s) to exercise
3  INFO: Exercising form http://localhost:3000/2009/7/16/hello-world |
4  //form[@id='comment-form']
5  WARN: Error code detected 500
6  INFO: Request invoked faulty reply
```

```
7| INFO: Found 1 invalid response
8| INFO: Created report in ~/report_Fri_Aug_28_17:03:33_+0200_2009.html
```

Listing 4.5: Fuzzing a form using YAFT

4.4.4 Identification and categorization of flaws

As you can see in listing 4.5, the fuzzing attempt invoked a faulty response indicated by the 500 status code. The next step is to see what caused the invalid response.

By inspecting the error log of the application-server that is listed in appendix C.1 . We see on line 29/31 that there was an uncatched exception caused by a stack overflow in the RedCloth library.

RedCloth[11] is a Ruby Libary for the markup language Textile[12] used by Mephisto to generate valid XHTML[13].

Our generated fuzz data (see lines 2 till 7 in appendix C.1) caused the parser in RedCloth to fall in a recursive loop that eventually resulted in a stack overflow. Since the server still sends a response we can categorize this flaw in *E2* (Failure to check return values).

4.5 Intelligent and Unintelligent fuzzing with YAFT

We explained how we generate attack scripts using YAFT and how we describe semi-valid fuzz data for web applications using a subset of the regular expression language.

We can use YAFT to fuzz web applications with random unintelligent fuzz data by defining the fuzz format specification for all form fields as `random`.

YAFT can intelligently fuzz Ruby on Rails web applications by extracting validators from the target application and translate them to fuzz format specifications as seen in earlier examples (see listing 4.1).

YAFT provides an additional method of intelligent fuzzing that we will call manually defining fuzz format specifications. This is simply manually

[11] RedCloth Textile for ruby: `http://redcloth.org/`

[12] The textile markup language: `http://textile.thresholdstate.com/`

[13] The Extensible HyperText Markup Language: `http://www.w3.org/TR/xhtml1/`

defining the fuzz format specifications of form fields in attack scripts.

Manually defining the fuzz format is an interactive trial and error process. First the fuzz format of a few form fields is specified and a fuzzing attempt is executed. By analyzing the response of the target application and inspecting log files, we can determine if the fuzzing attempt was hindered by: validators, the need for a specific format, or special values.

When this is the case, the fuzz format specifications are modified so that they generate data that has greater code coverage. This process is repeated as long as the application indicates that the processing of the form is stopped.

The manually defined specifications in the experiment performed in this thesis are trivial but require insight about the web applications from the tester. We eased this process by adding known valid values that are present in the form under test as comments in the attack script.

This process can be automated to an extend, however, the response of the target application is used to determine the fuzz format for following fuzzing attempts. The evaluation of the fuzzing responses are hard to automate and to our knowledge require human insight. Further research must be done to automate this process.

Chapter 5

Experiment

In order to evaluate our hypothesis that more flaws will be found using IF based on validators versus UF, we have used our fuzzing tool (YAFT) on a set of Ruby on Rails web applications that are presented in this chapter.

5.1 Test environment

The set of web applications that are tested in this experiment are a mix of modern open source and proprietary web applications, that use the Ruby on Rails framework. The proprietary web applications that are tested in our experiment are kindly provided and developed by Kabisa ICT[1]. Kabisa is one of the few companies in the Netherlands that specialize in web application development using Ruby on Rails.

In table 5.1 we give an overview of the web applications we will fuzz in this experiment. Column 2, 3 and 4 give some metrics such as the total lines of Ruby code (LOC) and the total number of classed and methods defined in the projects to give an indication of the complexity. Keep in mind that Ruby (on Rails) is a more efficient and condensed development platform than Java, for instance. Therefore, these metrics might skew your comparison image.

[1] Kabisa ICT: http://kabisa.nl/

web application	LOC	#Classes	#Methods	Description
Channelservice.fm	2237	33	243	Used by owners of streamit internet radios to manage their internet radio streams. It features a sophisticated administration backend for managing and auditing radios, streams and users.
Euroflorist	4140	95	290	Webshop framework for florists to create and manage their own web store that is available in several languages.
Lecturis	2806	57	286	Webshop for ordering on-demand print work.
Mephisto (v0.8)	4095	65	567	Opensource blogging system. Available at: `http://mephistoblog.com/`
Typo (v5.3)	10823	169	1178	Opensource blogging system. Available at: `http://wiki.github.com/fdv/typo/`

Table 5.1: web applications used in case study

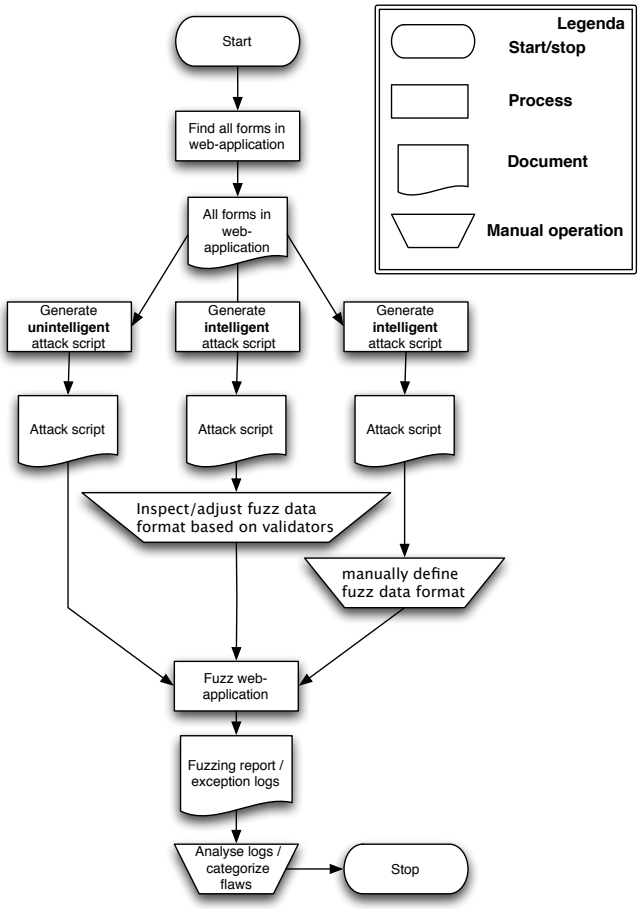

Figure 5.1: Experiment workflow

The workflow of this experiment is laid out in figure 5.1. First all the forms that are present in the web applications are detected using YAFT's form crawler. Then the detected forms are fuzzed using the three different methods that are described in chapter 4.5. Finally, the detected flaws are categorized as described in chapter 3.2 in order to measure if there is a difference in the kind of flaws that are detected using the different fuzzing methods.

web application	#Forms tested	#form fields	#Flaws UF	#Flaws IF
Channelservice	27	243	12	13
Euroflorist	16	66	1	1
Lecturis	16	78	6	7
Mephisto	18	120	4	5
Typo	19	160	4	5

Table 5.2: Fuzzing results

5.2 Results

Table 5.2 gives an overview of the results of this experiment. The first 2 columns show the number of forms and total number of form fields that are fuzzed. This excludes forms that were dynamically generated using Javascript/Ajax techniques since YAFTs form crawler has no support for javascript interpretation.

The third and fourth columns of table 5.2 give the number of flaws found using unintelligent fuzzing (UF) and validator based intelligent fuzzing (IF).

During the generation of the attack scripts for the IF experiment, there was still a great deal of manual interaction required. The test generator that YAFT provides, automated the translation from validator to fuzz data format specification for many of the validators used in the tested web applications. However, more than half of the form fields tested had multiple validators defined.YAFTs test generator is not intelligent enough to translate multiple validators to fuzz data format specification. In cases like that, the test generator will leave the fuzz data format empty for the tester to manually define but provides some help by adding the multiple validators as comments in the attack script.

This manual specification of fuzz data formats sparked our interest in fuzzing using manually defined fuzz data specifications. The web applications were fuzzed again but this time with manually defined fuzz format specifications. The results of this experiment are listed in table 5.3.

The attack scripts with manually defined fuzz data specifications are created by the author of this thesis, and are based on the automatically generated intelligent attack scripts.

web application	#Flaws UF	#Flaws IF	#Flaws manual fuzzing
Channelservice.fm	12	13	16
Euroflorist.fm	1	1	1
Lecturis	6	7	9
Mephisto	4	5	17
Typo	4	5	5

Table 5.3: Fuzzing results manual fuzz data specification

Fuzz method	#E1	#E2	#E3
Flaws UF	0	22	5
Flaws IF	0	26	5
Flaws manual fuzzing	0	42	6

Table 5.4: Detected flaws catogorized by method

The manually defined fuzz data specifications are trivial but required some insight in the workings of the target application. Examples for manually defined fuzz data specification are: using existing database IDs and specifying known valid values that are deduced from the target forms. As described in chapter 4.5 this is an interactive incremental process based on trial an error therefore it is hard to provide solid metrics for this part of the experiment.

And finally, we have the results of the categorization of the detected flaws in table 5.4. In this table we have grouped the categorized flaws by fuzzing method.

It is interesting to note that we did not find any occurrences of flaw category *E1* (resource exhaustion). However, we found multiple cases such as the example we give in chapter 4.4 where the application did yield a response but noticeably increased the load of the application server.

These kind of flaws can potentially be weaponized by sending multiple malicious requests in a short time span causing the server to overload and therefore become unable to respond to valid requests. This effectively is a denial of service attack (DoS)[2].

[2]Denial-of-service attack: `http://en.wikipedia.org/wiki/Denial-of-service_attack`

web application	Time UF	Time IF	Time manual fuzzing
Channelservice.fm	2:30	4:00	5:00
Euroflorist.fm	1:45	2:45	3:15
Lecturis	2:00	2:35	4:43
Mephisto	2:35	3:35	4:00
Typo	2:15	3:30	4:00

Table 5.5: Indication of time spent on fuzzing

5.3 Time indication

Table 5.5 gives a indication of the hours we spent on fuzzing each web application. This time includes the following activities:

- Generating attack scripts
- Manually adjust the fuzz format specification of attack scripts when needed.
- Fuzzing each form 100 times.
- Manually identifying and categorizing flaws.

This time excludes setting up the target applications and resetting the applications to a default state between UF, IF and manual fuzzing. A reset is also, needed in case a triggered flaw causes the whole application to stop functioning.

Note that we don't spend any time on isolating the fuzz data that caused flaws to trigger in order to replicate the bug without re-fuzzing. This is a logical step after fuzzing when you are actually attempting to fix the flaw (or exploit it).

In this thesis we have not focused on the performance of our fuzzer implementation. The speed of YAFT can be greatly improved by implementing some of the Ruby classes in C modules. The HTTP classes and random data generation implementations are great candidates for this. Also, some of the manual interaction that is required in our current implementation can be further automated. In chapter 7.1 these improvements are discussed.

The main bottleneck of fuzzing web applications is the response speed of the application under test. Too much concurrent requests can result in the

target application overloading, meaning that it can't process all the fuzzing requests, and this makes the fuzzing process even slower.

Another bottleneck is the amount of fuzz requests that are sent to the target application. As mentioned in chapter 4.3, we are currently sending 100 fuzzing attempts per test case. However, fewer requests with more diverse random data might be just as effective in triggering flaws. Further research can be done on this subject.

Chapter 6

Analysis and discussion

While the set of tested web applications were used in production environments, we still managed to detect 48 unique flaws using our 3 different fuzzing methods. And as expected, when fuzz testing, the flaws that were detected were not within the anticipated use cases of normal user behavior.

The experiment was set up to find out if more flaws are detected using our IF method that uses validators as a base for fuzz format specification. Using pure random UF, we managed to detect 27 flaws across the tested applications. Our IF method showed marginal improvement over UF detecting only 4 unique flaws extra while requiring substantially more manual interaction from the tester.

This manual interaction is mainly caused by tool weakness. The extraction of validators and translation to fuzz format specification has been automated for most of the common validators used in the tested web applications. However, when multiple validators are defined on one form field the translation to fuzz format specification has not be automated. More than half of the form fields fuzzed in this experiment had multiple validators defined; this caused the need for manual interaction. We leave the further automated translation of multiple validators for future work.

6.1 Flaw categorization

Of the total 48 discovered flaws, 42 (87.5%) are of the type E2 (Failure to check return values) meaning that many of the triggered flaws caused exceptions that were not gracefully handled causing the end-user to see unexpected error pages. These results indicate that return value checking of functions and exception handling are often overlooked in the set of tested web applications.

Since the number of unique flaws found using our IF method is very limited, we only found 4 additional unique flaws that were not triggered during UF. We can't make a comprehensive analysis about the differences of flaws found between IF and UF. However, we can say that for the extra 4 flaws that were detected using IF, the intelligent fuzz data indeed resulted in better code coverage passing the validators that were in place and triggering flaws that were not found with UF.

During our IF experiments we noticed that a substantial part of fuzzing attempts were ineffective and had low code coverage. These requests returned 404 HTTP status codes meaning that a resource cannot be be found. This was caused by the need for valid database references, such as database IDs that are found in hidden fields and select fields in the target forms.

When we manually defined the fuzz format specifications, many of these obstacles were bypassed and this resulted in triggering 17 more flaws than our IF method.

Our hypothesis is that if we make our current IF method smarter, by using known valid values that are extracted from the target application, the code coverage will be increased and more flaws will be triggered.

6.2 Threats to validity

A number of threats to validity exist with regards to this research. First off the number of web applications tested and flaws found are in no way comprehensive enough to give a basis for making bold statements about the effectiveness of IF on web applications. Testing a larger set of web applications with a wide diversity of functionality will yield more comprehensive data and a larger set of flaws to analyze.

Of the total 5 tested web applications 3 are developed by the same company and all the tested web applications are using the Ruby on Rails web-framework. This might give a bias to the quality of applications and flaws found. Also, web applications that are developed in other languages and frameworks might give different results.

For instance, In Ruby you are not forced to catch exceptions as is the case with Java. There might be a difference in flaws between web applications that are build in static languages and dynamic languages since many of the flaws we detected were caused by uncatched exceptions. Testing our IF method on a set open source web applications that are developed in other frameworks and languages will give a more diverse set of data to analyze and will allow us to make statements about the effectiveness of our IF method on web applications in general.

All of the experiments are performed by the same person including the manual specification of the fuzz data, flaw categorization and flaw analysis. This might have impacted the effectiveness of our method.

The effectiveness of the manual fuzzing method increases when the tester has more knowledge about the target application. The manually defined fuzz data specifications will have greater code coverage and will potentially trigger more flaws.

As the person who conducted this experiment doesn't know everything about the tested applications, we would expect the results to improve if the experiment ware conducted by testers who have in-depth knowledge about the target applications.

Chapter 7

Conclusion

Our work shows that fuzzing is a cost effective method for finding flaws in "tested and true" web applications but that intelligent fuzzing based on validators shows marginally better results, the drawback is that it requires more manual effort. This manual effort can be further automated, which would make it a valuable addition to fuzzing web applications.

7.1 Proposals for future work

During the process of this research we found several improvements that will potentially improve the effectiveness of our IF method. While our intelligent fuzzing method did not yield any spectacular results, it still managed to detect flaws that were not discovered using unintelligent fuzzing. The following recommendations might further improve and automate intelligent fuzzing of web applications.

- YAFT in it's current form is only able to fuzz forms. URL parameters, cookies and HTTP headers are also, viable vectors for UF.
- Fully automate the translation from validator to fuzz format specification even when multiple validators are used for a field.
- Support javascript evaluation in the form crawler in order to fuzz AJAX enabled sites.

- Use a proxy that captures outgoing HTTP requests and use the captured data to mutate semi-valid fuzz data.
- In order to get better code coverage, use valid values from hidden form fields (or other fields) as values in attack scripts, or extract valid values from the database of the target application.
- As mentioned in chapter 3.3, flaws do not always propagate immediately after a fuzzing attempt. We propose a solution for this problem by crawling the entire web application before a fuzzing attempt in order to check if pages that worked before are still working.
- YAFTs IF method currently only works with Ruby on Rails applications. Other web frameworks have similar validator functions that can be used to generate semi-valid fuzz data.
- In order to increase the performance of YAFT the HTTP request classes and random data generator classes can be implemented as C modules.

During our experiments we found that YAFT is a valuable tool in detecting input flaws in web applications. We plan to implement several of the suggestions we made in this chapter and release an open source solution in the near future.

Bibliography

[1] Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., and Vigna, G. Lncs 4176 - snooze: Toward a stateful network protocol fuzzer. 1–16.

[2] Cuadrado, J. S., and Molina, J. G. Building domain-specific languages for model-driven development. *IEEE SOFTWARE* (Aug 2007), 1–8.

[3] Fielding, R. T. Architectural styles and the design of network-based software architectures. 1–180.

[4] Forrester, J. E., and Miller, B. P. An empirical study of the robustness of windows nt applications using random testing. 1–10.

[5] Hammersland, R. Finding weaknesses in web applications through the means of fuzzing. 1–88.

[6] Hammersland, R., and Snekkenes, E. Fuzz testing of web applications. 1–6.

[7] Hieatt, E., Mee, R., and Evant. Going faster: Testing the web application. *IEEE SOFTWARE March/April 2002* (Feb 2002), 1–6.

[8] Howard, M., and Lipner, S. Inside the windows security push. *IEEE SECURITY & PRIVACY* (Jan 2003), 1–5.

[9] Howard, M., and Whittaker, J. Violating assumptions with fuzzing. *IEEE SECURITY &PRIVACY*, MARCH/APRIL (Mar 2005), 1–5.

[10] Keil, S., and Kolbitsch, C. Stateful fuzzing of wireless device drivers in an emulated environment. 1–11.

[11] Lipner, S., and Howard, M. The trustworthy computing security development lifecycle, March 2005. http://msdn.microsoft.com/en-us/library/ms995349.aspx.

[12] Luccaa, G. A. D., and Fasolinob, A. R. Testing web-based applications: The state of the art and future trends. *Information and Software Technology 48* (Aug 2006), 1–15.

[13] Maxwell, S. A. The bulletproof penguin, August 2001. http://home.pacbell.net/s-max/scott/bulletproof-penguin.html.

[14] Miller, B. P., Cooksey, G., and Moore, F. An empirical study of the robustness of macos applications using random testing. 1–9.

[15] Miller, B. P., Fredriksen, L., and So, B. An empirical study of the reliability of unix utilities. 1–22.

[16] Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A., and Steidl, J. Fuzz revisited: A re-examination of the reliability of unix utilities and services. 1–23.

[17] Miller, C., and Peterson, Z. N. J. Analysis of mutation and generation-based fuzzing. 1–7.

[18] Richardson, C. Orm in dynamic languages. *Commun. ACM 52*, 4 (Apr 2009), 48.

[19] Stuttard, D., and Pinto, M. The web application hacker's handbook. 1–771.

[20] Su, Z., and Wassermann, G. The essence of command injection attacks in web applications. 1–11.

[21] Xiao, S., Deng, L., Li, S., and Wang, X. Integrated tcp/ip protocol software testing for vulnerability detection. *Proceedings of the 2003 International Conference on Computer Networks and Mobile Computing* (Oct 2003), 1–9.

Appendix A

Pre-condition login script

```
1 @browser.get('http://localhost:3000/account/login') do |page|
2    login = page.form_with(:action => '/account/login') do |log|
3      log['login'] = 'admin'
4      log['password'] = 'test'
5    end.submit
6 end
```

Listing A.1: pre-condition login script

Appendix B

A generated attack script

```
1  TestForm ["http://localhost:3000/2009/7/16/hello-world","//form[@id='comment-form']"] do
2    name ""
3    action "/2009/7/16/hello-world/comments#comment-form"
4    method "post"
5    post_conditions :no_invalid_status_codes
6    cookie "c6d4360ed44b9c8a7d31924177d6b9121dcd713a\n"
7
8    fields do
9        input 'comment[author]', random #validates_presence_of :author
10       input 'comment[author_email]', /\w{1,10}@\w{1,10}\.\w{2}/ #validates_format_of :au
11       input 'comment[author_url]', random #""
12       input 'submit', random #"Submit Comment"
13       input 'comment[body]', random #""
14     end
15 end
```

Listing B.1: Generated attack script

Appendix C

Server logs during a successful fuzzing attempt

```
Processing MephistoController#dispatch (for 127.0.0.1 at 2009-08-28 17:03:27) [POST]
  Parameters: {"comment"=>{"author"=>"\243\201\016\215\a%p??\016\230\376   ?Y o",
  "body"=>"\227?{v\025???   \207\263\224\004? j3\231\254\234;z\00.\200}^u\271Q?y?",
  "author_email"=>"a43rjeir@vgh7fghg26frew.si",
  "author_url"=>"(X\2\271Q?y?"},
  "submit"=>"[\000?Z?\367",
  "action"=>"dispatch",
  "path"=>["2009", "7", "16", "hello-world", "comments"],
  "controller"=>"mephisto" }
  Site Columns (2.6ms)   SHOW FIELDS FROM `sites`
  Site Load (2.4ms)   SELECT * FROM `sites` WHERE (`sites`.`host` = 'localhost')
  LIMIT 1
  Site Load (0.2ms)   SELECT * FROM `sites` ORDER BY id LIMIT 1
  Article Load (0.5ms)   SELECT * FROM `contents` WHERE (`contents`.site_id = 1)
   AND ( (`contents`.`type` = 'Article') )
  Article Load (0.5ms)   SELECT * FROM `contents` WHERE (`contents`.site_id = 1
  AND ((contents.published_at IS NOT NULL AND contents.published_at <=
  '2009-08-28 15:03:27') AND (contents.published_at BETWEEN '2009-07-16 00:00:00'
   AND '2009-07-16 23:59:59') AND (contents.permalink = 'hello-world'))) AND (
  (`contents`.`type` = 'Article') )
  ORDER BY published_at desc LIMIT 1 Comment Columns (2.8ms)   SHOW FIELDS
  FROM `contents`
  Article Load (0.7ms)   SELECT * FROM `contents` WHERE (`contents`.`id` = 1)
  AND ( (`contents`.`type` = 'Article') )
  Site Load (0.3ms)   SELECT * FROM `sites` WHERE (`sites`.`id` = 1)
  SQL (0.1ms)   BEGIN
  SQL (0.2ms)   ROLLBACK
```

```
SystemStackError (stack level too deep):
    /vendor/gems/RedCloth-3.0.4/lib/redcloth.rb:998:in `glyphs_textile'
    /vendor/gems/RedCloth-3.0.4/lib/redcloth.rb:989:in `gsub!'
    ## shortened for this repeats for over 9000 lines!

Rendered /opt/local/lib/ruby/gems/1.8/gems/actionpack-2.2.2/lib/action_controller/
templates/rescues/_trace (5589.4ms)
Rendered /opt/local/lib/ruby/gems/1.8/gems/actionpack-2.2.2/lib/action_controller/
templates/rescues/_request_and_response (41.5ms)
Rendering /opt/local/lib/ruby/gems/1.8/gems/actionpack-2.2.2/lib/action_controller/
templates/rescues/layout.erb (internal_server_error)
loading from application: application_helper
  SQL (0.2ms)   SET NAMES 'utf8'
  SQL (0.1ms)   SET SQL_AUTO_IS_NULL=0
  Article Columns (3.0ms)   SHOW FIELDS FROM `contents`
  SQL (1.1ms)   SHOW TABLES
loading from application: mephisto_controller
loading from application: mephisto_helper
```

Listing C.1: Ruby on Rails log file (shortened for layout reasons)