ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Laura Peer

# Fuzzy Attacks on Web Logs

Group Thesis (GA)-2011-09
August 2011 to September 2011

Tutor: David Gugelmann
Co-Tutor: Dr. Stephan Neuhaus
Supervisor: Prof. Dr. Bernhard Plattner

## Abstract

Oftentimes, weaknesses in web applications arise because input is not properly validated. Web programmers can make mistakes or may simply not conceive of every possible input scenario provided to their application.
In this thesis we use the method of fuzzing, the automated process of feeding pseudo random input data to a program, to test web applications for bugs. Our target applications are web logs. With the help of a small program which was written for our specific task, we have discovered a number of problems and vulnerabilities.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

A report issued by the National Institute of Standards & Technology in 2002 [1] states that the annual cost estimates of an inadequate infrastructure for software testing is estimated to be $59.5 billion. Heuristically, there are 5 to 10 bugs per 1000 lines of code still remaining after initial product testing [2]. The earlier a defect is detected, the less costly it will ultimately be to fix.
The ever-growing size and integration of the Internet into our daily life and the fact that a continuously increasing amount of private and sensitive data is being put into online storage, stresses the need for developers to be very alert and considerate of security threats, even at the very beginning of their projects.
A good example to illustrate this issue is the attack on Gawker Media in 2010 [3]. It took a group of attackers less than 24 hours to gain access to the full source code of the site and the entire database containing millions of passwords and email addresses. According to the attackers, the security measures used by Gawker were severely outdated. To make matters worse it turned out that the stored passwords were not as strongly encrypted as they should have been with the terrible result that around 1.3 million entries could be decoded using brute force within hours.
This and countless other similar events show that sometimes companies do not prioritize security as much as they should with devastating results.
When software users entrust their sensitive data to companies and services which handle this privileged information so carelessly their choice of even the strongest and most varied passwords will be rendered moot. There clearly exists an utmost and growing need for good security measures in the programming phase as well as later in software testing.
While there are many approaches that test for security integrity, the concept of fuzzing is a powerful tactic due to its ease of implementation and history of great results.

## 1.2 The Task

This thesis focuses on how well web logs handle user input. The method we employ to generate the user input is called fuzzing.
Fuzzing is a means of testing software for possible vulnerabilities. We therefore attempt to generate abnormal behavior of various web log software by supplying either completely random data, or data that has been heuristically proven to potentially cause unavailability or ultimately erroneous behavior, possibly leading to security issues.
For this purpose we set out to write a program that specifically targets web logs, by first ascertaining what inputs the specific website asks for and then proceeding to generate fuzzed strings and feeding them to the website. The resulting behavior of the websites, given the altered data, is then detected by our program and is further analyzed and interpreted.

## 1.3 Related Work

The term “fuzzing” was coined in 1988 by Barton Miller, a lecturer at the University of Wisconsin-Madison. One evening, he noticed that the UNIX applications, which had active remote connections crashed frequently during a thunderstorm [4]. He concluded that the reason for the abnormality was due to the noise on the link, no doubt produced by the heavy rain.
While it was obvious that noise could distort transmitted messages, it surprised him that the garbled input on the link caused the programs to malfunction. In order to investigate these curious events, he suggested the issue of robustness of various UNIX utilities be analyzed in form of student projects. It was at that stage that he came up with the name “fuzzing”. As Barton Miller himself states in the foreword to his upcoming book [4] he was looking for a word that evoked the feeling of random, unstructured data.
Those first attempts at fuzzing were purely geared towards improving the reliability of UNIX applications. The early testers had not yet placed their focus on system security analysis. There was strong criticism placed on fuzzing in the software testing community in the early stages of development, no doubt due to the primitive nature of stuffing random bits brute-force into application inputs. The results, however, soon started speaking for themselves.
Developers and testers alike moved their focus from simple UNIX performance testing to a broader set of goals. A great number of programs have since been developed, including comprehensive fuzzing frameworks like PeachFuzzer [5] and SPIKE [6].
Fuzzing frameworks provide libraries and functions with integrated fuzz data generation and target behavior monitoring which facilitate the fuzzing of a target. Those frameworks absolved interested users from writing their own code and deriving their own methods of testing. They were broad enough to cater to a multitude of different targets. This approach can be advantageous for some purposes. Generally though, when a fuzzing program for a specific task is needed, the testers are better off developing their own software.
Pedram Amimi held a presentation on fuzzing frameworks on July 28, 2007 [7] wherein he stated that there is a steep learning curve involved when setting out to work with some of the bigger frameworks, due to their sheer size and complexity. Additionally, since most of these frameworks have been around for quite a while, the results of a test might prove fruitless. Most developers will have already fixed the bugs uncovered by the framework algorithms in the meantime.
The increasing popularity of fuzzing as a testing method led even the bigger developers to start writing their own private fuzzing software to test their products before release. Google [8], Microsoft [9] and Mozilla [10] are three examples of big companies that created their own fuzzing programs for their software testing phase.
Some programs conduct local fuzzing. They focus on testing applications for possible stack and integer overflows in command-line input and a broad range of other vulnerabilities which may arise when opening malicious files. Local fuzzing tends to place bigger importance on reliability and stability of software and less on security, since we generally do not expect security threats to live in our own backyard.
However, the massive growth of the Internet altered the shape of computing. While this held enormous benefits considering ease of access and global communication for the corporate as well as the private sector, it unfortunately opened the doors for a whole new set of then unforeseen security threats.
The consumers want to be able to access everything from the newest movie to their private banking endeavors in the blink of an eye. Especially the latter prospect needs to be handled with the utmost reliability and protection in place. Since this calls for software to be error proof and secure, the time for testing remote applications in terms of security vulnerabilities was at hand.
This fact is illustrated by a number of fuzzing programs geared towards remote targets like DNS, email, web applications, web browsers and web servers. Protocols define the means for remote communication and data transfer. Because of this, a number of remote fuzzing programs and frameworks started attaching themselves on knowledge of a specific network protocol like HTTP, SMTP and FTP. For example the Firefox JavaScript fuzzer jsfunfuzz [11].

## 1.4 Overview

Chapter 2 outlines the theory behind our three main concepts. Those entail the method of fuzzing, the means of transport of packets between a web server and how web applications gather input through HTML forms.
In Section 2.1 we delve into a description of the method of fuzzing, first taking a look at the chronological process, then explaining the available data generation techniques. Section 2.1.3 ventures into a description of various vulnerabilities that can be detected when employing the fuzzing of a target.
Basic knowledge on how to build a fuzzing program alone would be useless if we did not have detailed knowledge of the lower-level means of data transfer between the server and web browser. Therefore, we take a closer look at the HTTP protocol in Section 2.2.1 by analyzing a simple request for a website.
For us to understand regular and irregular server behavior, we describe the HTTP communication process between server and browser in Section 2.2.2, while Section 2.3 contains a brief introduction to HTML forms and shows how they can be sent to a web application.
Chapter 3 shows the inner workings of the program we wrote for our task. Section 3.1 handles a precursor to our main routine, the task of crawling the website for information about which input fields to attack. In Section 3.2 we illustrate how we designed our program's network interface. Section 3.3 details our programs data generation method. In Section 3.4 we introduce our programs backbone, the main routine and in Section 3.5 we show how we handled error detection both on the client and server system.
Chapter 4 details the results we obtained after running our program against three different versions of blog software.
Finally, we proceed to give a brief summary and outlook in Chapter 5.

# Chapter 2

# Background

## 2.1 Fuzzing

### 2.1.1 The Fuzzing Process

A tester first sets out on such a project by selecting the specific application he wants to test. After the selection process, he can proceed to analyze the application interface.
As soon as he is aware of the way the selected application receives its input, he can start generating fuzzed data by either building chunks of random data or modifying preexisting inputs to the target software randomly.
Once the data is ready to be fed to the application, the tester can automate and execute a process that sends the fuzzed inputs to the program.
Equally important as having good data generation is the detection of faults generated by the bad input. If the tester does succeed in provoking an error or unexpected condition in the software it becomes vital for his task's success that he locates the exact piece of data responsible for the anomaly. Saving all outgoing packets in sequence is therefore a must, otherwise he would need to backtrack through thousands of packets while looking for the culprit.
One approach could be communicating with the application once after every attempt to send fuzzed data and seeing how long it takes to respond. Delays and irresponsiveness could then be indicators for that last packet being successful in causing an unexpected condition.
A second approach may be glancing at system and application log files and scan those for error messages and other abnormal entries, since some bad inputs may trigger the application to output error messages.
The best and most comprehensive way of analyzing software behavior is accomplished by using a debugger, where the environment allows it. This can easily be done if one attempts to fuzz locally. By using this approach it is possible to catch some vulnerabilities that were not caught before, though they did trigger an error condition, perhaps because the application had some error handling routine restoring it to a good state very cleanly and quickly and without the tester's detection mechanism being alerted.
Finally, after having triggered and detected possible anomalies in execution, comes the complex task of analyzing the results and developing a sense on which vulnerability may lie behind which error, software crash or slowdown.

### 2.1.2 Random Data Generation

The approach of preparing fuzzed data can be broadly categorized as either generation-based or mutation-based [12]. The former approach creates random data from thin air, the latter approach modifies valid, preexisting software input data. Especially in the latter case, there are more and less complex ways of doing so.
A more sophisticated procedure can detect more specific and harder to find bugs a lot faster than very simple and rudimentary random data generators, but even they get there in the end. This fact illuminates the beauty and simplicity of fuzzing.
The most primitive fuzzing programs fill the entire input with random data without knowing what results that method will bring forth in the end. This technique reminds of a brute-force approach

and might generally be regarded as ineffective, but has nevertheless yielded some stunning results in the past.
Fuzzing is relatively simple to implement and allows a preliminary screening, because it can shed light on really awful code [12]. Unfortunately, it is a resource waster. Most applications scan their input according to a specific formatting pattern, perhaps even including a checksum prior to accepting it as an input, and would just drop most of the fuzzed data because it does not correspond to the expected syntax and formatting.
Router software for example only accepts TCP/IP packets according to strict formatting rules (destination address, source address, length field, checksum and so on). It would be a waste of resources to engage in complete randomization of the data packets in this specific case.
The previous point illuminates the fact that the more sophisticated the approach of data creation is, the more knowledge about the targeted software needs to be gathered as a precursor. That entails looking at patterns in the input and deciding which parts to randomize and which parts to preserve for a successful acceptance into the application.
Instead of purely randomized clusters of data, attack heuristics can and should be used to obtain ultimate test results. Attack heuristics are methods developed by studying previous vulnerabilities which occurred on a variety of systems and in differing contexts.
We can compare this to an experienced, knowledgeable tester sitting in front of a web browser inspecting a site for an SQL injection vulnerability. Hopefully his solution does not include bashing multiple keys randomly on his keyboard in the hopes of exploiting the vulnerability. Instead he would attempt to send a well thought-out and specific request to the server (something along the lines of `"OR 'a'='a"`, see Section 2.1.3, SQL Injection).
Attack heuristics, in short, teach the fuzzing program to proceed like an experienced tester or attacker would, attempting to achieve what worked well in the past. This is necessary because we do not have infinite time and resources in testing. If we had, we could simply run all the possible permutations of our complete symbol-set, thus, in the end covering all heuristics-produced data as well as countless meaningless, garbled and non-result-wielding messages.
The different vulnerabilities discussed in the next Section serve as the crux for smart data creation, specifically when a more sophisticated web application fuzzing client is to be developed.

### 2.1.3 Vulnerabilities

The study on the subject of fuzzing has led us to notice countless examples of weak-points in software. They hold great importance when developing attack heuristics for a network application fuzzer.
In this section we describe those vulnerabilities that were predominantly found in web applications in the past, while omitting the ones that would only pertain to file corruptions and various other local applications and more specific protocols.

#### Integer Overflows

Data storage in computers is not infinite. The task of a processor consists mainly of arithmetic operations. Since the processor registers contain the inputs and outputs of arithmetic operations, and those registers are typically between 8 and 128 bits wide, there exists a maximal number for every system.
A machine that uses 32-bit registers can represent values up to $2^{32} - 1$, which equals 4’294’967’295. Architectures can use half of this scope to represent negative numbers. Looking at an 8-bit field, the numbers would range from -128 to 127 instead of 0 to 255.
The danger in that representation arises if we add 1 to the highest number (or subtract 1 from the smallest). The result is said to wrap around, jumping from the highest positive number to the smallest negative number (127+1=-128, in the 8-bit case).
This in itself is not a problem, however when software expects a positive value and a user enters a negative one, or a negative one is achieved by such a wrapping event, unexpected conditions may occur.
When designing fuzzed data, it can thus be helpful to generate very big numbers or negative numbers to test if the programmer of the software put the proper safeguards into place.
Additionally one should test a number of boundary cases consisting of numbers close in value to the maximal number and perhaps even numbers close in value to the maximal number

divided by 2, 3, 4 and so on.

### Buffer Overflows

When a program is executed, the system allocates a region of memory, where instructions and other program specific data like variables and attached libraries are stored.
Programs are dynamic in nature. They grow and contract with time. The construct which defines how the program is stored in memory is called the program stack.
Some programs deal with strings or character arrays. They allocate buffers or arrays of characters with previous knowledge on the intended target size of the data.

```
char array[12]; # allocates a 12 byte character array
```

They may additionally expect the user to input text into the program. Some routines that handle those strings of characters do not check for the size of the provided data. Examples are `gets`, `scanf` and `strcpy`. If those routines are used in combination with a string that exceeds the intended length of the previously created array, the program stack gets overwritten beyond the boundaries set by the array, causing program data to be overwritten, including the return address which is at the bottom of the stack.
It must be noted that compilers catch boundary mismatches like this one

```
array[i]='c';    # i>=arraysize
```

This means that the issue of buffer overflow only arises if a program accepts additional input from the user at runtime, and that the user generates more data than the programmer intended him to provide.
For security reasons, the operating system restricts writeable areas to the scope of the program in memory. If a malicious user inputs a very large string, the program will crash throwing a segmentation fault, because the input data will flow over the boundaries of the program stack, attempting to write data to restricted areas in memory.
If an attacker or tester sees a segmentation fault after inputting large strings, he can assume that the program is vulnerable to buffer overflows. He could then start deducing the size of the vulnerable buffer through guesswork, with a debugger, or by dumping the crashed image to disk and rifling through its contents.
Once the size is determined, he can proceed to fill that buffer with meaningless characters and add as a trailing addition a portion of very specific data, perhaps a starting address of another program, perhaps a shell. If the program he exploited is setuid, meaning that it has the ability to run on elevated privileges to do certain tasks, the spawned shell may be granted root access to the machine. It is up to the programmer to use those routines that employ boundary testing, especially when the program asks for user-generated inputs.
A fuzzing program should be able to flood every input that takes a string with thousands of characters in order to detect whether the software is susceptible to buffer overflows.

### Format String Variables

Format string vulnerabilities most commonly arise in C when a program receives user input which is not properly validated. The format string scheme tells the compiler how it should format numbers when it sets out to print them.
There are a number of C functions that accept format string arguments. For example `printf`, `fprintf`, `sprint`, `vprintf` and `vfprintf`.
A sample call could look like this

```
printf("number: %d.\n",42); #output: "number: 42."
```

Notice that `printf` takes two arguments. The first one is a string containing the format parameter `%d`, the second one is a number. The format parameter tells the function what kind of data the second argument is and additionally, where it should be placed in the final string.
If the programmer omits the format parameter, the routine simply takes what it needs from the program stack, which the main reason for the existence of the string format vulnerability.

Table 2.1 specifies some interesting format parameters.

| Format Parameter | Type of Data | Passed As |
|---|---|---|
| %d | decimal (int) | value |
| %x | hexadecimal (unsigned int) | value |
| %s | string (const unsigned char*) | reference |
| %n | number of bytes written (int*) | reference |

Table 2.1: Format String Examples

An example of a problematic command is

```
printf("%s%s%s%s%s%s%s%s%s%s")
```

`Printf` causes the format parameter `%s` to read the values from the program stack up to the point where it reaches an illegal address (non-program memory) and gets terminated by the operating system, causing denial of service (DoS). While this is a nuisance, there are other user-supplied strings that can cause more damage.
For example supplying `printf` with no format parameters, but with a string that contains a number of `%x` causes it to show the contents of memory in hexadecimal format. This allows an attacker to view at least parts of the program stack, which is not good, since it can facilitate his quest for more serious exploits.
Using `%n`, a smart attacker can write data of arbitrary size to arbitrary locations in the program stack, potentially causing buffer overflows and altered return addresses, leading him to obtain root access rights.
When building a good fuzzing program it is appropriate to use combinations of `%s` and `%n` as inputs.

**SQL Injection**

Many web applications use an SQL database backend to structure some aspects of their site. Blog softwares, for example, generally write their contents to a database and e-commerce platforms want some form of storage system that provides for billing information, client addresses and order forms.
A regular SQL query when checking for the existence of a username within a database could look similar to this statement

```
SELECT * FROM database WHERE username = 'stringentered'
```

As a starting point, an attacker could simply insert a "’" symbol somewhere and see what happens.

```
SELECT * FROM database WHERE username = 'some'thing'
```

If the target escapes the received string, it will simply not find the user in the database. However, if input protection is off at the target, the last input will result in an SQL syntax error. The attacker can now deduce that SQL injection will most likely be possible in this setup and can proceed to add statements similar to

```
some_input OR 'a'='a
```

This will cause the WHERE clause to be true in any case, because the AND operation is stronger in binding than OR in computer logic.

```
SELECT * FROM database WHERE some_category='something' AND
    other_category='some_input' OR 'a'='a'
```

Will become equivalent to

```
SELECT * FROM database
```

This very basic example marks an approach that enables an attacker to bypass the authentication scheme of the SQL database.
If the software performs no validation of inputs, an attacker could perform his own unauthorized queries on the database, inject data, perhaps add an additional user into the database and tamper with files in the worst case.
It is useful, when designing a remote fuzzing program, to have it produce symbols commonly used in SQL queries. This little additional effort can achieve to trigger SQL injection hazards.

### Cross Site Scripting

Cross Site Scripting arises when an attacker succeeds in implanting some of his own code into a website. When a website gets infected, every future visitor will download and execute the implanted code along with the normal intended HTML source code, not doubting the authenticity of the data, and that, in turn, can cause the visitor's system to perform a potentially malicious action. Web services as diverse as Hotmail, Orkut, Myspace and even Paypal have been hacked using XSS exploits [13].
The injected code could be written in any scripting language. It makes sense here, however, to choose amongst the more popular scripting languages like JavaScript or HTML, which can be understood by all web browsers.
Maliciously injected code could be intended to perform a phishing attack of broad scale or it could simply collect cookie information. In some cases the altered website could even show a completely new interface that tricks users into giving it login credentials.
The smart thing about this attack is that users and their machines trust those infected hosts implicitly, because they are thought to be harmless and secure.
The protection mechanisms against Cross Site Scripting should lie within the web application itself. It is vitally important that user generated input gets properly sanitized. The application inputs should be closely scanned for any identifiers of an executable script language, for example `(<,%,>,?)`.
A fuzzing program can be taught to enter various different attack vectors into a target web application. Afterwards it only remains to be seen if the web page, when refreshed, reflects any changes in its HTML source code, specifically whether it will contain the entire unescaped injection. If it does, it is indeed vulnerable to Cross Site Scripting.

### Directory Traversal

An attacker could attain unauthorized read and possibly write access for the entire server file system if that system is vulnerable to directory traversal.
Web browsers often display web addresses like these

```
http://example.com/file_opener.php?home=file.txt
http://example.com/file_opener.asp?page=file.txt
http://example.com/file_opener.jsp?file=file.txt
```

The `*.php`, `*.asp` and `*.jsp` routines serve as file openers. Usually, those methods present the files to the visitor in a nice and usable format.
The tag names `"home="` ,`"page="` and `"file="` are used by those respective routines to denote a path to a file.
This command

```
http://example.com/file_opener.php?home=/path/to/file/found.txt
```

could open up the found.txt file on a vulnerable server, provided that we know the full path to the file and the input is not sanitized.

One specific file of interest on Linux servers would be the /etc/passwd file. It holds the passwords for the server system users. On Windows the C:\WINDOWS\system32\cmd.exe would be of interest.
The path to the parent directory is denoted by "..", thus a request for ../../example.txt will try to open up the file example.txt that is located two levels above our current position in the file system tree structure (..\.. \in Windows).
Today, most systems store password hashes for their users in the /etc/shadow file, which is only read-accessible with administrator rights. Nevertheless, it might be interesting for an attacker to at least find out which users are registered on the system, which is still deducible by reading the /etc/passwd file.
To do so, he could proceed with the following query

```
http://example.com/file_opener.php?file=../../../etc/passwd
```

There may be some minor brute force required when determining how many "../" to add to the path, since we cannot know for sure how far down we are in the file system tree, but especially by using automated fuzzing tools this proves to be a small inconvenience.
When fuzz testing for Directory Traversal vulnerabilities, the tester should teach his fuzzing program to insert strings of different amounts of appended "../" or "..\" combined with existing filenames.
The following encodings are additionally useful for testing for all possible omissions in security measures

```
%2e -> .
%2f -> /
%5c -> \
```

After the fuzzing program throws the generated data at the web server the detection routines should know what to expect when those files are in fact opened as the tester wished. Scanning the reply page source code for known fragments can inform about success.

## 2.2 Fuzzing the HTTP Protocol

The goal of our thesis is attacking web log software. Before we can set out to develop data generation and error detection routines for our fuzzing program, we need to inspect the internal workings and rules set forth by the underlying HTTP protocol. This knowledge prooves vital in order to recognize which input fields exist and how we could alter and randomize them.
Our goal is to change only so much that the server will still accept the packet as a valid HTTP request.

### 2.2.1 Client to Server Transmission Analysis

Let us see what happens behind the scenes, when a web browser is told to open a URL. The following output was obtained by capturing the network traffic using a packet sniffer tool like wireshark [14] and request a URL using our browser.
After an initial DNS retrieval process and TCP/IP handshakes our web browser has successfully found the IP address of the target web server. It then proceeds to send an HTTP packet to the server like this one

```
GET / HTTP/1.1
Host: www.somehost.com
User-Agent: Mozilla/5.0 Safari/533.21.1 Accept: text/plain
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: "received from server"
Connection: keep-alive [\r\n\r\n]
```

Line 1 specifies the request method “GET”, a path “/” and the HTTP protocol version 1.1. The range of accepted methods varies from server to server.
Generally a server will accept a minimum of three known methods. "HEAD", "POST" and "GET". "HEAD" requests only ask to retrieve the header information from the server. The methods "GET" and "POST" are similar.
If a user fills an input form on a webpage, for example posting an entry to a blog, he wants the browser to send that data to the application. The browser can proceed to send that data using either the "GET" or "POST" method. The data is then appended to the end of the HTTP packet as a payload in a "POST" request like this

```
entrytitle=firstentry&body=firstbody
```

or appended to the destination address when using the "GET" method like this

```
http://someblog.com/path/to/entry.php?title=firstentrye&body=firstbody
```

In both schemes, the different data entries are separated from each other using the ‘&’ symbol. Specifically in "GET" they are appended to web address using the ‘?’ symbol as an additional separator.
As we can see, when using the "GET" method to send data, the full contents are visible in the browser URL, which can be an unwanted side effect.
Additionally, the "POST" method allows for more data to be sent. This issue arises because, for example, web proxies want to be able to store the entire URL, meaning that it couldn’t grow enormously big depending on how much data needs to be sent, while in a "POST" request the payload simply gets sent in chunks to the same URL.
It is up to the application programmer to determine which method he expects and allows for which input form.
Notice that the “/” path on line 1 of our example request is called the Uniform Resource Identifier (URI). Here our browser requested the base website, since we did not enter anything other than the base address into the browser’s URL field. If we wanted to access a page like this

```
http://www.ee.ethz.ch/en/our-range/education/bachelor.html
```

the host would be

```
http://www.ee.ethz.ch/
```

and the path would become

```
/en/our-range/education/bachelor.html
```

Our browser sent additional fields, lines 3-8, called headers, along with the request. These headers are name-value pairs and carry additional and optional information that pertains to the client browser itself, the requested page, the server and more. The host header was made mandatory with the introduction of HTTP version 1.1.
A header could for example describe the file formats the browser understands, or it could specify the kind of operating system and browser software which was used in making the request. This line is especially useful if any formatting differences arise across the different platforms and software.
There are a number of accepted HTTP headers in use. The comprehensive list of valid header names is listed in chapter 14 of the Hypertext Transfer Protocol Reference [15].
A tester who focuses on HTTP fuzzing would attempt to alter all possible fields of an HTTP request like the one seen before. That would entail setting large and small integers as the version number, providing random input as the header names and values, the host, the URI and even the method portion of the packet. The purpose of this approach is to see whether the server scans and protects itself enough from bad HTTP packets.
If the main goal of a tester lies in analyzing web applications and not the underlying server, it is more useful and appropriate for him to provide the server with valid HTTP packets, wherein he

only changes those values which will serve as web application inputs. These values are sent as the payload of the "POST" request, or, if the "GET" method was used, appended to the URI.
In addition to the data there is one specific header that is of interest, when fuzzing web applications. The cookie header seen on line 7. It is generated by the web application running on the server and in contrary to the other headers not created by the server itself.
Since HTTP is a stateless protocol, it does not inherently have means of keeping connections open, nor does it strive to, since that is exactly the purpose of the cookie, which enables the web application to maintain certain state information. This can be anything from authentication to user preferences.
When receiving a valid cookie, the server recognizes the identity of the client and can thus show saved shopping baskets or it can grant authorized access to an area of the site.

### 2.2.2 Server Response and Detection Methods

When setting out to develop detection mechanisms for fuzzing programs it is important to understand the format of server-generated replies. We need to know what a normal reply looks like in order to distinguish it from an erroneous one.
The web server answers a "GET" or "POST" request by sending us back an HTTP packet similar to this

```
 1  HTTP/1.1 200 OK
 2  Server: nginx
 3  Date: Mon, 29 Aug 2011 12:59:22 GMT
 4  Content-Type: text/html; charset=UTF-8
 5  Transfer-Encoding: chunked
 6  Connection: close
 7  Vary: Accept-Encoding
 8  Expires: Wed, 11 Jan 1984 05:00:00 GMT
 9  Last-Modified: Mon, 29 Aug 2011 12:59:22 GMT
10  Cache-Control: no-cache, must-revalidate, max-age=0
11  Pragma: no-cache
12  Link: <http://wp.me/1NEo1>; rel=shortlink
13  Content-Encoding: gzip [\r\n\r\n]
14  <html>
15      #page source in HTML with possible scripts in other languages
16  </html>
```

On line 1, the server echoes the requested HTTP version 1.1 and returns a status code number. The number 200 in this example stands for a successful retrieval of the requested website. See chapter 10 of the Hypertext Transfer Protocol Reference [15] for the complete list of possible status codes.
Lines 2 to 13 show the server's reply headers.
Finally, from line 14 on we observe the queried website's source code. This portion is appended in the payload of the servers reply. It is additionally the only portion of the reply our web browser will make visible to the end user.
In order to detect a possible error condition when fuzzing a tester should compare the received status codes.
Imagine our client attempting Directory Traversal and succeeding after a number of attempts. The responses of the server could be several status code 404: Not Found errors, followed perhaps by a status code 403: Forbidden status code.
In other cases a web application can embed error messages in its source code. As we have seen in SQL injection the attacker usually deduces a possible vulnerability from an error message he triggered. This error message can be displayed on the website. It can easily be filtered out by inspecting the response payload that consists of the HTML source code of the requested website, therefore it might prove useful for tester to continually request the same website, each time scanning the source code for changes.
If the web server crashes, communication with the client is ceased. Due to this fact, it might not be a bad idea to periodically test for responsiveness and to determine whether the response

time has gotten bigger due to some unexpected behavior, or if the sever has indeed crashed due to some bad input.
Servers and web applications each maintain error log files, which should be inspected for clues in real time during the testing phase.
Finally, for the most comprehensive results in analyzing and detecting faulty behavior, a tester could run the web application in a debugger, although, since this process cannot be automated easily, this method applies to more specific attacks that are being performed by hand and is not suited for an approach that sends multiple packets to the web application in a short period of time.

## 2.3 Web Application Input

Web applications can be seen as a collection of webpages that seamlessly form a fully functioning program. The pages of the application form a tree structure. Some applications may even contain a subtree that is only accessible with login credentials. In a blog software this would be the maintenance section of the blog, where the administrator can manage the blog and write or delete entries.
Some pages contain links to other pages in the tree structure, some others are end nodes. If a browser requests a page from the tree, the server sends its HTML source code as a payload appended to the HTTP reply packet.
This source code contains formatting rules for the page data as well as information about embedded links and fill-out forms, in addition to the destination URL and the request method.
If we now look at the way the HTML code is structured we can gather that the form fields specify exactly which inputs the application expects. Here is an example form field

```
1  <form action=" http :// foo . org / form . php " method=" get ">
2    <input name='username ' type="text" value=""
3    <input name="email" type="text" value="">
4  </form>
```

Line 1 tells us that this form needs to be sent to the destination address specified in the action field, using the HTTP request method "GET".
Lines 2 and 3 specify the form data variables that are expected by the application. In this case, the data would be appended to the URI like this

```
http://foo.org/form.php?username=exampleuser&email=foo@bar.org
```

HTML adheres to a set of formatting rules. This makes it easy to automate the process of gathering all possible application inputs using a web crawler, for example.
Ideally, such a web crawler traverses the entire website tree and collects all form variables. Those variables can then be forwarded to the fuzzing program which assigns them random values and sends them back to the web application.

# Chapter 3

# Design

In this chapter we will describe the structure and methods of our fuzzing program.
We have decided against using a preexisting framework, mainly because we were more interested in learning the underlying method of fuzzing, rather than learning what methods the framework contained and how it was to be used correctly.
We wrote our application in Python. This choice was supported by Python's comprehensive selection of nice libraries and methods, which enable and simplify network communication as well as Python's support for neat data structures like dictionaries, which came in handy for our data throughput.
Finally, it did not hurt that we were particularly fond of the language structure from the start.
Figures 3.1 and 3.2 illustrate our program's running process and its data dependencies.

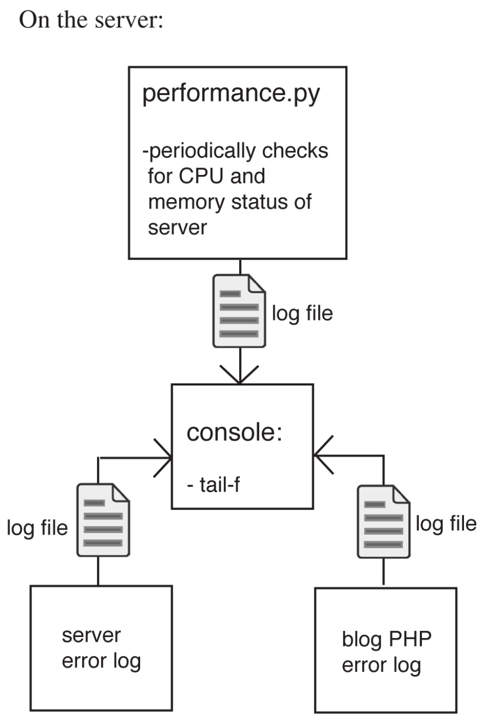


Figure 3.1: Flow Chart of our Fuzzing Program on the Server

On the attacking machine:

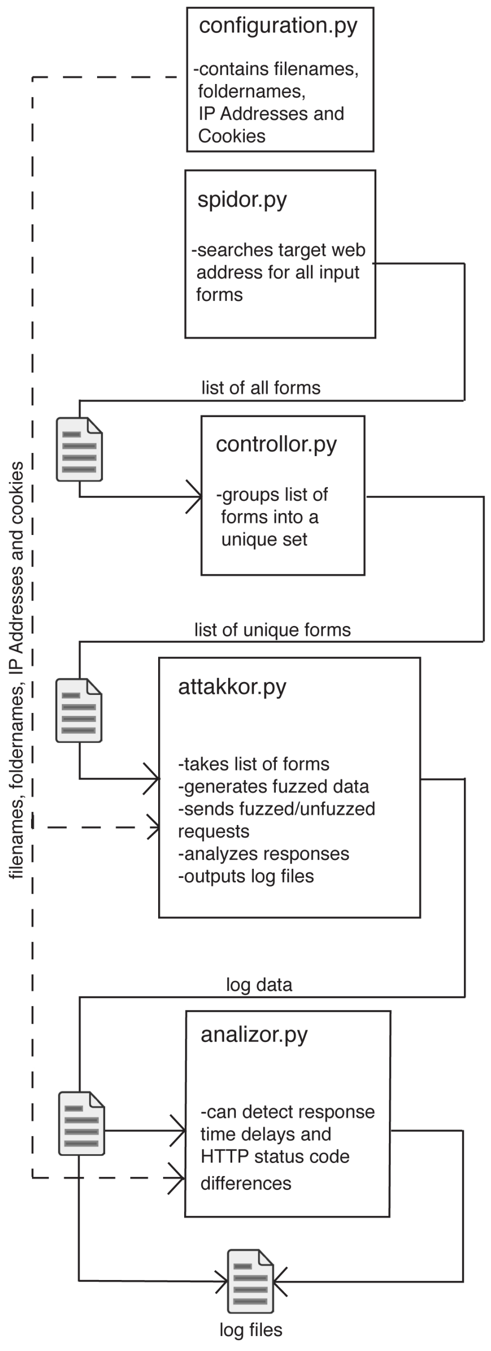

configuration.py
-contains filenames, foldernames, IP Addresses and Cookies
spidor.py
-searches target web address for all input forms
list of all forms
controllor.py
-groups list of forms into a unique set
list of unique forms
attakkor.py
-takes list of forms
-generates fuzzed data
-sends fuzzed/unfuzzed requests
-analyzes responses
-outputs log files
log data
analizor.py
-can detect response time delays and HTTP status code differences
log files
filenames, foldernames, IP Addresses and cookies

Figure 3.2: Flow Chart of our Fuzzing Program on the Attacking Client

## 3.1 Analyzing the Web Log with a Web Crawler

The initial step before fuzzing was finding the application inputs. In order to accomplish this, we used a small web crawler, which a friend and fellow student, Stephan Müller, wrote and kindly shared with us.
His web crawler accepts a URL as an argument and promptly starts traversing the web application tree from that node on extracting the form data from the HTML code on every node.
Finally, it outputs a file which contains every unique and application-approved input form.

## 3.2 The HTTP Communication Interface

Python offers a variety of networking support schemes of differing abstraction level. The lowest approach would be to send a request to the server by opening a network socket and simply feeding it the entire HTTP request as a string.

```
import socket
socket.setdefaulttimeout(20)
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
s.send(request)
buffer=""
# wait for the entire datastream to be captured
while 1:
    response=s.recv(1024)
    if not response:break
    buffer+=response
s.close()
```

This approach can be useful especially for server fuzzing, since we generate the entire HTTP request string by hand which gives us complete control over what data is sent, enabling us to fuzz all the fields we want and as much as we want.
We soon realized that it was sufficient for our specific task to use the simpler and more abstract methods of urllib2 for our network requests.
The following section of code illustrates a HTTP "POST" request using the methods described in urllib2.

```
1 import urllib2
2 request=urllib2.Request(url,payload)
3 response=urllib2.urlopen(request)
4 server\_status\_code=resp.getcode()
```

On line 2 we generate a request in HTTP format by invoking the `Request()` method of urllib2.
On line 3 we use the `urlopen()` method to send the newly formed request to the destination of our choice.
We chose this method over the socket based approach, because our fuzzing target is the web application and it was not in our interest to show standalone server vulnerabilities. We were only interested in fuzzing the portions of the HTTP packet specifically pertaining to the web application inputs, which are the cookie and the payload.
If we want access to the privileged section of the web application we need to teach our program how to accept and store cookies from the application.
This could be done like this

```
import urllib
url="http://somesite.com/login.php"
opener=urllib.URLopener()
payload={'name':'loginname','pw':'password'}
response=opener.open(url,urllib.urlencode(payload))
```

```
6 cookies=opener.info()['set-cookie']
```

On line 5 we sent login name and password in the payload to the `login.php` routine of the web application. The application contained cookies in its response header, which we filtered out by using the dictionary key identifier `'set-cookie'`, which is the standard format for cookie data according to HTTP.
Since HTTP is a stateless protocol we needed to include this cookie in any subsequent request, if we indeed wanted to gain access to the web application's privileged section.
The following code shows how we can append a cookie to the header of an HTTP request using

```
opener.addheaders.append(('Cookie', cookie))
```

## 3.3 Fuzzed Data Generation

The most primitive approach, when generating data, is to simply create strings of random symbol sequences of varying length like this

```
1 fuzz_string=""
2 string_length=random.randrange(1,limit,1)
3 selection=
4     'abcdefghijklmnopqrstuvwxyz0123456789/\\%$#_.:!-*()|\'<>,?&+=@";'
5 selection='/\\%$#_.:!-*()|\'<>,?&+=@";'
6 for i in range(string_length):
7     fuzz_string=random.choice(selection)
```

Line 2 sets the length of the generated string to a random number, while line 6 keeps appending a random symbol from the string called `selection` until the set length is reached. We have started out with the first selection string on line 4, however, we soon realized while fuzzing the web application that it was sufficient for us to just fuzz using the special symbols seen on line 5. Those special characters are regularly used in a number of languages, like % for a format string or ' in SQL for example and are multiple degrees more likely to cause problems than the letters and numbers.
If we wanted to generate a buffer overflow attack, we could simply enter a very big value for the variable `limit` on line 2, creating a very long string.

## 3.4 The Main Program Loop

### 3.4.1 Working with the Web Sniffer Output

Our program's main routine is in charge of tying all components together. It starts by reading in all forms, which were generated by the web sniffer. An example form looks like this

```
{
"action": "http://192.168.1.70/wp-comments-post.php",
"data": [["comment", "", "textarea"],
         ["comment\_post\_ID", "3", "hidden"],
         ["submit", "Submit Comment", "submit"]],
"method": "post"
}
```

This form is sent to the application, whenever a comment is to be posted on the blog. The action field tells us that the input is sent to the `wp-comments-post.php` routine. The request method is HTTP "POST", which specifies that the payload needs to be appended after the headers of the sent packet. The data key tells us which input variables the application is to be given.

### 3.4.2 The FuzzField Class

The form we saw in the previous subsection contains a method, an action and a data field. We are only interested in fuzzing the values in the data field, since those represent the inputs which the application is going to process.
A form can have multiple data fields. For ease of detection, we have decided to only fuzz one data key at a time. Had we not done that, there would be no way for us to find out which field caused the error.
Our program consists of a loop which circles through all data entries, placing a value into a different field on each turn. We decided to pass the field we want to fuzz by reference. We accomplished this by assigning an instance of a class we created and called FuzzField to the data key in question. This approach is advantageous, because it allows us to alter the contents of the fuzzfield in the input form on the fly and in every part of the program.
The FuzzField class looks like this

```
 1  class FuzzField(object):
 2      """reset value: instance.set(value); print: str(instance)"""
 3      def __init__(self, arg):
 4          super(FuzzField, self).__init__()
 5          self.value = arg
 6      def set(self, value):
 7          self.value = value
 8      def __str__(self):
 9          return str(self.value)
10      def __add__(self,other):
11          return str(self)+str(other)
```

Line 5 shows how we can assign a value to an existing object by invoking `instance.set(value)`, while line 7 enables it to print its internal value, when using the `str(.)` directive.

## 3.5 Error Detection Mechanisms

### 3.5.1 Detection Methods while Fuzzing

Fuzzing is done by automation and is thus very fast paced. To detect errors if they occur we needed to implement detection schemes that worked hand in hand with the program we built.
For information clarity we made our program generate unique and incrementing sequence numbers for every attempt of sending a fuzzed input string to the application. We wrote the sequence numbers and random strings into a file, thus gaining the ability to repeat the entire attack at a later time, if we wanted.
Additionally, our program wrote one main log file. Our testing runs produced mostly useless data. Any interesting event that does not cause a complete server crash will most likely not be caught by eye, so we decided to write some useful real time statistics into the aforementioned log file, which was formatted in a special way, allowing us to search for all occurrences of specific events within the log file after we were done attacking the server. In order to replay a complete attack scenario, the file was provided with the following items

```
sequence number, fuzzed value, current time stamp, complete input
form field, response time, server status code and Difflib string
similarity ratio, inserted, equal and deleted symbols
```

The `fuzzed value` and the `complete input form field` are required to perform the same query on the server in a replay scenario.
The `current time stamp` was useful, when we wanted to compare it to some event in the server error log file. Even though the clocks were not exactly in sync, there would only remain a small interval of insecurity when determining the culprit value.

The `response time` is the time difference between the sending of the fuzzed HTTP request and the retrieval of the response from the server. Knowing approximately how much time a normal request needs until it is handled, allowed us to monitor server behavior and helped us detect any potential lags in performance.
We decided to send one initial unfuzzed packet before we entered the loop that generated a set number of fuzzed ones, even though this approach generates a little bit more traffic. The reason we did that was to be able to compare all the potentially unhealthy responses with the initial healthy one. This allows us to compare the two `server status codes`.
As you recall from Section 2.2.2, the server sends a `status code` on the first line of its response packet. If the request was fruitful it returns the `status code` 200. However, if the server fails to fulfill a seemingly valid request, we may see a 5xx `status code`. For an error on the client's side, we get a 4xx code. See the chapter 14 of the Hypertext Transfer Protocol Reference [15] for the comprehensive list of `server status codes`.
In our detection approach we decided to look for `server status codes` that were specifically in the 5xx range and 4xx range, since that usually meant that something went wrong.
Since we had two responses from the server, a healthy one, and a fuzzed one, we could use Python's `Difflib` library to compare the differences in the two HTML page sources. `Difflib` contains a method called `SequenceMatcher` that can calculate the number of symbols, which have been `deleted`, `replaced` or `inserted`. The `ratio` column lists a floating point number, which measures the similarity of the websites. Python's `Difflib` Documentation [16] states that a ratio of 0.6 and higher is an indicator that the strings are quite similar.

```
1 s=difflib.SequenceMatcher(lambda x: x in " \n\t\r",html1,html2)
2 inserted=0
3 for tag, i1,i2,j1,j2 in s.get_opcodes():
4     if tag=="insert":
5         inserted+=j2-j1
```

Line 1 of this code creates an instance of the `SequenceMatcher` object which compares the old HTML contents with the new ones. The `SequenceMatcher` method uses a slightly altered version of the "gestalt pattern matching" algorithm developed in the late 1980s by Ratcliff and Obershelp [17]. The first argument is a junk indicator telling it to ignore empty spaces, newlines, carriage returns and tabs.
On line 3 we used the `get_opcodes()` method, which returns a 5-tuple. This tuple's job is to describe how to turn the first string into the second by giving us the respective position in the strings where the difference started, `i1` and `j1`, and the end of the different portions `i2` and `j2`. The first entry of the tuple is the identifier, which tells us how the strings differ, if they were `replaced`, `deleted` or textttinserted. In the code above, we have only calculated the number of `inserted` characters. Our program does the same thing for the number of `deleted` and `equal` symbols.
The significance of this comparison varies greatly among all the different input forms. If we look at a search form, for example, the results of this `SequenceMatcher` comparison will produce great differences, since we first search for an empty string which will output the entire blog contents on the page and then proceeds to search for our fuzzed string which is most likely not in the blog, yielding no results.
However, if a program has an obvious SQL injection vulnerability in some input field, and we suddenly find a small descriptive error on the website after entering some fuzzed input, our `SequenceMatcher` approach will most likely generate a smaller number of `inserted` characters and otherwise many equal ones. Either way it can be useful to keep those numbers in mind, and have an idea of the expected number patterns, when fuzzing the application.
In addition to the log file and for additional completeness of detection, we wanted to store the server response HTML source code of both fuzzed and unfuzzed requests.
To save space, we have made our program dump those HTML files into a `.tar.gz` archive. They were given the name of the current `sequence number`, so we could proceed to unzip a single file from the archive by just knowing the `sequence number` in question, if we discover something interesting in the log output for example.
Finally, we decided to search the fuzzed HTML responses for keywords using regular expressions. For obvious SQL vulnerabilities it might make sense to search the HTML page for the

keywords "error" and "SQL". In some applications with directory traversal vulnerabilities, it may make sense to search for a string along the lines of "File not found". Since we already stored all the HTML responses in an archive, we only needed to write out the `sequence number` of the matching document.

### 3.5.2 Server Side Detection

The server and blog software both maintain separate log files that may hold important run-time error messages. Before every fuzzing run we opened up the console and tailed the logfiles.
In addition to the logs, we wrote a small script that we ran on the server during run-time. Its main task is to collect status information of all threads that come from requests of our fuzzing program.
This program regularly searches the server's `server-status` module for all process ids that originate from the fuzzing program by matching the IP address of the attacking client. A sample entry from the `server-status` page looks like this

| Srv | PID | Acc | M | CPU | Req | Conn | Child | Slot | Client | VHost | Request |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2-0 | 1964 | 0/1/1 | - | 0.05 | 58 | 12.9 | 0.02 | 0.02 | attacker-ip | localhost | GET /blog.php?p=14 |

After we have gathered a list of all involved process ids, we invoke `ps aux`. For each line in the `ps aux` output we match the process id to the ones we have previously collected. If there is a match, we sum up the values for memory and CPU percentage as well as real and virtual memory size and write the results to a file.
This file is also tailed in a shell before running the fuzzing program. It lets us see in real time if the server memory or CPU get overloaded at any time, while the attack process is taking place.

# Chapter 4

# Results

## 4.1 Experiment Setup

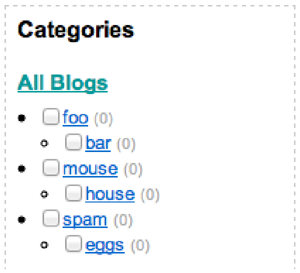
We have decided to run the server in a virtual machine. This allowed us to set memory snapshots of good server states. The great advantage of this approach was that we could revert the whole system to the last known good state within seconds. If any data was corrupted in the process, the inflicted damage would only be contained to that virtual partition.
Furthermore, it made sense to virtualize the network packet transmissions, since we wanted to send many requests in short succession and a real network would have had caused unnecessary delays.
We set up the server on a Ubuntu/10.04 distribution running Apache/2.2.14 with PHP/5.4.3 and MYSQL/5.1.41 installed. Since blog applications put their contents into databases, we have created a separate database for the blog and a user with specific privileges using phpmyadmin/3.3.2.
Initially, we decided to test the functionality of our fuzzing program on old software, preferably one that was in its development infancy.
We chose a lesser known open source blog platform called b2evolution/0.8.6 [18], developed by François Planque. The version 0.8.6, which we installed dates back to September 2003.
After, we decided to test WordPress [19], one of the most commonly used open source blog platforms with and without plugins.

## 4.2 Testing b2evolution/0.8.6

Our program did not have to run a long time against b2evolution to find issues. Within seconds we saw multiple response pages, which had matched for the search term "SQL". An example string that caused such an error is

```
"(>&-\;-)\>@/(|.@.!@\,#/.=,.&!;*$(?*+@*<#.|;\<')+'-!/&?&>(."
```

A quick inspection of the source codes in question showed that we had indeed found multiple SQL injection vulnerabilities. In each of those vulnerable forms the application did not escape the input at all before placing it into the SQL database query. To make matters worse it generated an error page when the issue was triggered. This allowed our program to find the vulnerability within seconds.
Finally and awfully, the error message displayed the complete SQL database query string, making it extremely easy for an attacker to figure out a possible attack input, which could allow him to tamper with the database. Even the simple input of the symbol ' caused the error message to be displayed.
b2evolution provides the means of assigning blog entries to various categories. On the right side of the main blog page, the user is given the option to select categories of interest through a checkbox, see Figure 4.1.


Categories

All Blogs

- foo (0)
  - bar (0)
- mouse (0)
  - house (0)
- spam (0)
  - eggs (0)

Figure 4.1: b2evolution/0.8.6 Category Select Field

### 4.2.1 The catsel[] Field

When he submits his selection, the page will refresh and only show entries from the selected categories. In a normal scenario this form takes its input solely from the mouse which checks the wanted boxes and leaves unwanted ones empty, finally, proceeding to click submit. There is no field in that input form where the user could enter a string. This lead the programmers to, very mistakenly, not prepare for bad input, even though it can be very easily generated by just looking at the HTML source code and extracting and replaying the form field in question which is displayed in Figure 4.2.

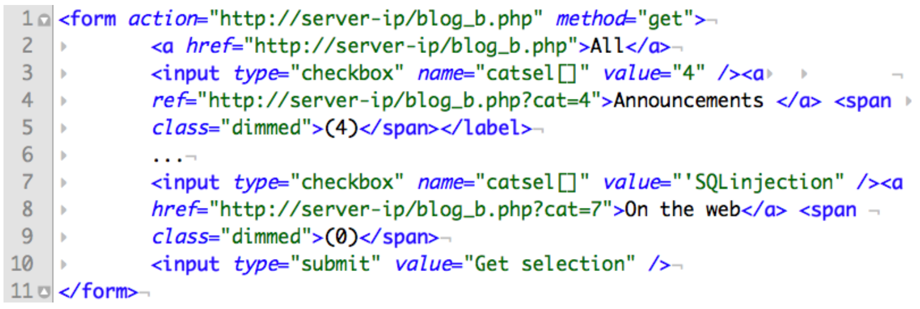

```
<form action="http://server-ip/blog_b.php" method="get">
        <a href="http://server-ip/blog_b.php">All</a>
        <input type="checkbox" name="catsel[]" value="4" /><a
        ref="http://server-ip/blog_b.php?cat=4">Announcements </a> <span
        class="dimmed">(4)</span></label>
        ...
        <input type="checkbox" name="catsel[]" value="'SQLinjection" /><a
        href="http://server-ip/blog_b.php?cat=7">On the web</a> <span
        class="dimmed">(0)</span>
        <input type="submit" value="Get selection" />
</form>
```

Figure 4.2: b2evolution/0.8.6 Category Select Form Source Code

When assigning the `catsel[]` field of Figure 4.2 a string like 'SQLinjection, which we can do by using the "GET" method, we can produce the following SQL error message

```
Oops, MySQL error!
Your query:
SELECT DISTINCT ID, post_author, post_date, post_status, post_lang, post_content, post_title,
post_trackbacks, post_category, post_autobr, post_flags, post_wordcount, post_karma FROM (evo_posts
INNER JOIN evo_postcats ON ID = postcat_post_ID) INNER JOIN evo_categories ON postcat_cat_ID
= cat_ID WHERE cat_blog_ID = 3 AND postcat_cat_ID IN ('SQLinjection ) AND ( post_status IN
('published') ) AND post_date <= '2011-09-22 22:50:50' ORDER BY post_date DESC LIMIT 3
MySQL said:
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near 'published') ) AND post_date <= '2011-09-22 22:50:50'
ORDER BY post_date DESC LI' at line 1
```

In addition to the category selector form in the previous section we found the search form of the blog to be vulnerable to SQL injections as well. Since the search form is sent with the HTTP method "GET", an attacker could easily play around with the following URL

```
http://server_address/blog.php?s='SQLinjection&sentence=AND&submit=Search
```

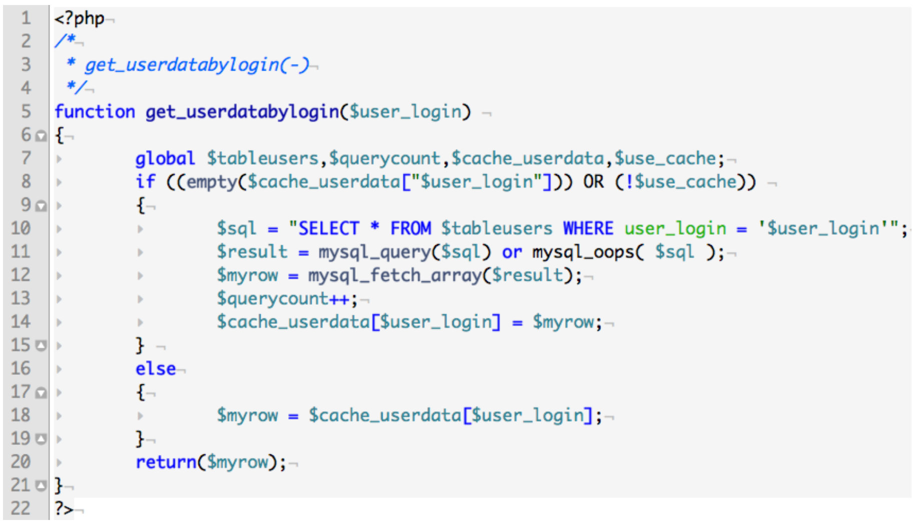
By merely changing the value assigned to the `"s"` key in the above URL we can send query after query very easily. In contrary to the category selector form in Section 4.2.1 the programmer must anticipate the user to enter typed strings into the search field. This SQL vulnerability is very

easily triggered.
A user might stumble upon it without ever even looking for it, since an accidental stray ’ in a search string will generate the following error message

```
Oops, MySQL error!
Your query:
SELECT DISTINCT ID, post_author, post_date, post_status, post_lang, post_content, post_title,
post_trackbacks, post_category, post_autobr, post_flags, post_wordcount, post_karma FROM (evo_posts
INNER JOIN evo_postcats ON ID = postcat_post_ID) INNER JOIN evo_categories ON postcat_cat_ID =
cat_ID WHERE 1 AND ( ( (post_title LIKE ’% ’SQLinjection %’) OR (post_content LIKE ’%’%’) ) ) AND (
post_status IN (’published’) ) AND post_date <= ’2011-09-23 00:30:07’ ORDER BY post_date DESC LIMIT
3
MySQL said:
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near ’%’) OR (post_content LIKE ’%’%’) ) ) AND ( post_status
IN (’published’) ) AND ’ at line 1
```

### 4.2.2 The Login Field

A third issue we stumbled upon, when analyzing b2evolution/0.8.6, is another SQL injection vulnerability. This time, the blog application did not properly handle the login name value it received from the login form. Figure 4.3 shows the PHP source code of the function called `get_userdatabylogin()`, which the blog application uses to query the database for the existence of the user within the database.
Notice that the variable `$user_login` contains user entered string and is not escaped by the function. The routine pieces together the pre-formed SQL query by appending this unescaped variable to the existing query string on line 10 in Figure 4.3.



```
1  <?php
2  /*
3   * get_userdatabylogin(-)
4   */
5  function get_userdatabylogin($user_login)
6  {
7          global $tableusers,$querycount,$cache_userdata,$use_cache;
8          if ((empty($cache_userdata["$user_login"])) OR (!$use_cache))
9          {
10                 $sql = "SELECT * FROM $tableusers WHERE user_login = '$user_login'";
11                 $result = mysql_query($sql) or mysql_oops( $sql );
12                 $myrow = mysql_fetch_array($result);
13                 $querycount++;
14                 $cache_userdata[$user_login] = $myrow;
15         }
16         else
17         {
18                 $myrow = $cache_userdata[$user_login];
19         }
20         return($myrow);
21 }
22 ?>
```

Figure 4.3: b2evolution/0.8.6 Login Handler Routine

### 4.2.3 The Search Field

When entering the example string ’SQLinjection into the user name field, we see the following SQL error

```
Oops, MySQL error!
Your query:
SELECT * FROM evo_users WHERE user_login = ’’SQLinjection ’
MySQL said:
```

```
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server
version for the right syntax to use near ''SQLinjection ' at line 1
```

These examples and their implications illustrate how important it is to escape user input. Even if the user isn't actually expected to type anything, the specific values can still be collected from the page source code, altered and sent away in the background.
b2evolution/0.8.6 has gaping security flaws, especially when we consider that those vulnerabilities are accessible from the public section of the blog. Anyone with no privileges whatsoever could perform operations on the database, which are of the same privilege status as is granted to the blog application. That entails for example altering the contents of the blog or deleting posts or comments.
b2evolution/0.8.6 dates back to 2003 and is the very first version listed on the main product website [18] in the "Release History" section, where the author explicitly states that this release is unsafe to use, since it was never a release candidate. Because of this we are not going to send a bug report to the developers. Nevertheless, it served as a great initial testing subject and enabled us to show that fuzzing can indeed be a very helpful tool for detecting very bad coding practices.

## 4.3 Testing WordPress

The following section outlines our trial and error process of testing WordPress with our fuzzing program.
Online research into the history of WordPress lead us to cvedetails.com [20], a security website, which carries a comprehensive section solely devoted to gathering and categorizing security holes within WordPress software. It clearly states the corresponding threat level, the vulnerability category, the version number and the year the vulnerability was discovered in.
Based on this information we initially chose to install and test a WordPress/2.0 release. According to cvedetails.com, from which we extracted the values in Table 4.1, WordPress/2.0 contains the most vulnerabilities.

| Version | Number of Vulnerabilities |
|---|---|
| 2.0 | 46 |
| 2.0.1 | 41 |
| 1.5 | 40 |
| 1.2 | 38 |
| 1.5.1 | 38 |
| 1.5.1.2 | 38 |
| 2.0.4 | 38 |

Table 4.1: Number of Vulnerabilities detected per Version of WordPress

### 4.3.1 WordPress/2.0 and 3.2.1

Having completed the steps of installing the blog on the server and filling it with some posts, we were ready to run our fuzzing program against WordPress/2.0. We generated a little over 3 million requests within four days of continuous testing.
During the entire testing period, there was not a single occurrence of denial of service (DoS). The response times we measured never increased and there were no significant CPU bursts. The server did not experience any cumulative growth in CPU and memory load during testing. Our analysis of the apache2 error log file did not reveal any abnormalities, neither did the PHP error log file that we instructed Wordpress/2.0 to use for debugging messages. Additionally, our keyword search of the response HTML yielded no results.

Even though we initially chose this version because of its big number of vulnerabilities, we did not detect any of them with our fuzzing program. It was time to explore other options, so we installed WordPress/3.2.1, the current version as of July 12th 2011.
After running our fuzzing program against WordPress/3.2.1 for two days totalling in over 1 million recorded requests, it had again failed to find any vulnerabilities in the blog application.

### 4.3.2 WordPress/3.2.1 with Plugins

Our next approach was to install a number of plugins and repeat the tests. We chose to install the top 5 plugins listed in WordPress's plugin directory [21], in the "Most Popular" category. This entailed Google XML Sitemaps/3.2.6, Contact Form 7/3.0, All in One SEO Pack/1.6.13.4 and 1 Flash Gallery/1.6.2.
It took some initial preparation to tell the web crawler where to look for the plugin input forms, since some of them were not linked on the main blog pages.
After 10 hours of testing and more than 500'000 generated requests our search for vulnerability indicators still remained fruitless.

### 4.3.3 The WordPress Administrative Section

Listed in cvedetails.com [20] are a number of vulnerabilities which can only be exploited within the administrative section of the respective WordPress software. Because of this, it made great sense to additionally test this part of the blog software, both in version 2.0 and 3.2.1.
For us to successfully access privileged pages, our fuzzing program and the web crawler both needed a newly implemented method, which, upon start, retrieved the session cookie from the blog application and appended it to every outgoing packet header.

#### The Wordpress/2.0 Administrative Section

While looking at the cookies WordPress/2.0 provided us, we stumbled upon a security issue. The cookies we got from Wordpress/2.0 stayed equal across subsequent sessions. This opens the blog software up to replay attacks.
A websniffer could easily extract the cookie from an intercepted packet header and replay it at any time in the future, thus gaining unwarranted administrative access to the blog application.
The corresponding bug report was filed by Steven J. Murdoch on November 19, 2007 [22]. This, however, remained the only breach of security we found when fuzzing the administrative section of version 2.0.

#### The Wordpress/3.2.1 Administrative Section

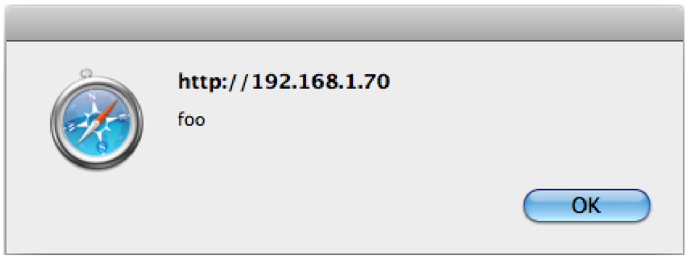
When we told the web crawler to search the WordPress/3.2.1 administrative section for forms, it was processing for more than one hour without coming to a conclusion. This made us suspicious and we cancelled the operation. After some investigation we found that it had ordered the download of over 300 plugins and 30 themes and it would have kept going almost indefinitely, had we not stopped it. However, a quick escape clause in the web crawler code fixed that issue.
In our subsequent tests, we found that the fuzzing program, while neither crashing nor lagging, had created a number of new posts on the blog and altered some menu entries in the administrative section. A subsequent run of the web crawler unexpectedly yielded no available forms.
After some analysis, we found that a blog entry, created by our fuzzing program, had corrupted the HTML source code of the main blog page, leading the web crawler to ignore the main page and not being able to move on due to the lack of further hyperlinks.
Looking at the main page we quickly found the culprit post.
This let us think that it might be possible to inject code into the "add new post" form of the administrative section. To prove that javascript was injectable, we logged in and created a new post with the title

```
<script>alert("foo")</script>Post Title
```

and in the body we wrote

```
<script>alert("bar")</script>Post Body
```

When we now attempted to open the blog page, we first got a JavaScript alert telling us "foo", see Figure 4.4, then a subsequent one, telling us "bar", proving that code can indeed be injected into a new post title and body fields. After the alerts were clicked away, the new post simply read "PostTitle: Post Body". While this is only possible with administrative privileges, this issue still affects any user who visits the blog that sports such a post.


http://192.168.1.70

foo

Figure 4.4: JavaScript Alert on the Main Blog Page

Attempting to insert the same code in a new post on another blog, which was hosted on WordPress.com, did not work. We were unable to ascertain, whether this issue was recently fixed on their site, or if it only ever pertained to version 3.2.1 of the self-hosted blog software.
Some online research led us to find the corresponding bug report, filed by Darshit Ashara on August 28, 2011 [23], which describes the issue we found in sufficient detail.

### 4.3.4 WordPress/3.2.1 with a Vulnerable Plugin

Our last attempt at testing WordPress/3.2.1 entailed installing a plugin which contains a known vulnerability. We thought that exact knowledge of the location of the vulnerability may help us produce additional results.
We decided to install oQey gallery/0.4.8 [24] to this effect, which contains an SQL injection vulnerability. Its bug report was filed by Miroslav Stampar on September 5, 2011 [25]. Within the report he states that the method called `getimages.php` of the oQey plugin does not properly escape and sanitize its input of the variable `$gal_id`.
The input form of the `getimages.php` routine uses the HTTP "GET" method to provide the value of the id key like this

```
http://site.com/wp-content/plugins/oqey-gallery/getimages.php?gal_id=<injected code>
```

An attacker can append a specific attack string to the end of this web address, causing a SQL database query of his choosing to be executed.
Since we now knew which input form specifically contained the inputs for the `getimages.php` method, we could start automating the process of feeding it random values using our fuzzing program and hoping for promising results. Here, however, is where we got stuck. The plugin does not use any alert mechanisms. If any SQL syntax errors were to occur, they would be handled silently, without a displayed error message. This stands in contrary to the results of testing b2evolution/0.8.6 in Section 4.2, where the error messages alerted us to the SQL syntax error in the first place.
Vulnerabilities like the one just mentioned are generally hard if not impossible to detect using an automated technique of fuzzing. They are rather found, by sifting through the application source code, while looking for sections of code, where form input is not escaped.

Chapter 5

# Outlook and Summary

The main goal of this project was searching blog applications for vulnerabilities.
To accomplish this, we wrote a fuzz testing program in Python which is contained in the attached CD-ROM.
Taking HTML input form parameters, the program allows us to generate and send random requests to a blog application without violating the rules of the HTTP standard.
It contains mechanisms for detecting abnormal conditions in the blog application. Among them are testing the request response times, comparing HTTP status codes and searching the received responses for keywords.
We have tested b2evolution/0.8.6 and found multiple SQL injection vulnerabilities with ease. We found a security flaw in the way WordPress/2.0 handles cookies. In WordPress/3.2.1 we detected a Cross Site Scripting vulnerability in the privileged section of the blog.
The program could further be improved by incorporating heuristic attack vectors that target specific vulnerabilities. For this addition to be complete, the detection methods should be improved with further knowledge of the desired effect of the attack vectors on the blog application.
An example of an additional method of detection which could be implemented is using a website parser to analyze the HTML integrity of every response received by the blog application in order to detect potential script injection vulnerabilities.
The fuzzing process is cheap and very fast to implement compared to other, more sophisticated testing methods. It is a good addition to regular code checking.

# Bibliography

[1] National Institute of Standards and Technology. Software Errors Cost U.S. Economy $59.5 Billion Annually. http://web.archive.org/web/20090610052743/http://www.nist.gov/public_affairs/releases/n02-10.htm, June 2002. Last visited: 2011-09-25.

[2] Humphrey, W. S. Bugs or Defects? http://www.sei.cmu.edu/library/abstracts/news-at-sei/wattsmar99.cfm, April 1999. Last visited: 2011-09-25.

[3] Halliday, J. and Arthur, C. How a million users' details got gawked at. http://www.tmcnet.com/usubmit/2010/12/14/5192174.htm, December 2010. Last visited: 2011-09-25.

[4] Miller, B. Foreword for Fuzz Testing Book. http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html, March 2009. Last visited: 2011-09-25.

[5] Eddington, M. Peach Fuzzing Platform. http://peachfuzzer.com, June 2006. Last visited: 2011-09-25.

[6] D. Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 2002.

[7] Amini, P. Fuzzing Frameworks. http://www.blackhat.com/presentations/bh-usa-07/Amini_and_Portnoy/Whitepaper/bh-usa-07-amini_and_portnoy-WP.pdf, July 2007. Last visited: 2011-09-25.

[8] Anantharaju, S. Automating web application security testing. http://googleonlinesecurity.blogspot.com/2007/07/automating-web-application-security.html, July 2007. Last visited: 2011-09-25.

[9] Fuzz Testing at Microsoft and the Triage Process. http://blogs.msdn.com/b/sdl/archive/2007/09/20/fuzz-testing-at-microsoft-and-the-triage-process.aspx, September 2007. Last visited: 2011-09-25.

[10] Ruderman, J. JavaScript fuzzer available. http://blog.mozilla.com/security/2007/08/02/javascript-fuzzer-available, August 2007. Last visited: 2011-09-25.

[11] Rudermann, J. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz, August 2007. Last visited: 2011-09-25.

[12] Sutton, M. and Greene, A. and Amini, P. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, first edition, 2007.

[13] Anderson, R. J. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, second edition, 2008.

[14] Combs, G. Wireshark, the world's foremost network protocol analyzer. `http://www.wireshark.org`. Last visited: 2011-09-25.

[15] Gettys, J. and Mogul, J. and Frystyk, H. and Masinter, L. and Leach, P. and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, June 1999. Last visited: 2011-09-25.

[16] Python Software Foundation. 7.4 difflib - Helpers for computing deltas. `http://docs.python.org/library/difflib.html`. Last visited: 2011-09-25.

[17] Ratcliff, J. W. and Metzener, D. Pattern Matching: The Gestalt Approach. *Dr. Dobb's Journal*, page 46, July 1988.

[18] Planque, F. b2evolution: More Than A Blog! `http://b2evolution.net`. Last visited: 2011-09-25.

[19] WordPress.org. `http://wordpress.org`. Last visited: 2011-09-25.

[20] Wordpress: Vulnerability Statistics. `http://www.cvedetails.com/vendor/2337/Wordpress.html`. Last visited: 2011-09-25.

[21] WordPress Plugins » Most Popular. `http://wordpress.org/extend/plugins/browse/popular`. Last visited: 2011-09-25.

[22] Murdoch, S. J. Wordpress Cookie Authentication Vulnerability. `http://www.cl.cam.ac.uk/~sjm217/advisories/wordpress-cookie-auth.txt`, November 2007. Last visited: 2011-09-25.

[23] Ahara, D. Wordpress 3.2.1 Core Module Improper Sanitizing. `http://www.exploit-id.com/web-applications/wordpress-3-2-1-core-modulepost-template-php-improper-sanitizing-xss`, August 2011. Last visited: 2011-09-25.

[24] oQey Gallery. `http://oqeysites.com/oqey-flash-gallery-plugin`. Last visited: 2011-09-25.

[25] Stampar, M. WordPress oQey Gallery plugin <= 0.4.8 SQL Injection Vulnerability. `http://unconciousmind.blogspot.com/2011/09/wordpress-oqey-gallery-plugin-048-sql.html`, September 2011. Last visited: 2011-09-25.