Finding weaknesses in web applications through the means of fuzzing

Rune Hammersland

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2008

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Finding weaknesses in web applications through the means of fuzzing

Rune Hammersland

2008-06-30

Abstract

The handling of input in web applications has many times proven to be a hard task, and have time and time again lead to weaknesses in the applications. In particular, due to the dynamics of a web application, the generation of test data for each new version of the application must be cheap and simple. Furthermore, it is infeasible to carry out an exhaustive test of possible inputs to the application. Thus, a certain subspace of all possible tests must be selected. Leaving test data selection to the programmers may be unwise, as programmers may only test the input they know they can expect.

In this thesis, we describe a method and tool for (semi) automatic generation of pseudo random test data (also known as “fuzzing”). Our test method and toolkit have been applied to several popular open source products, and our study shows that from the perspective of the human tester, our approach to testing is quick, easy and effective. Using our method and tool we have discovered problems and bugs with several of the applications tested.

An article version of the thesis is included in Appendix C.

Keywords: D.2.5.k Testing strategies, D.2.5.o Test execution, D.2.5.r Testing tools.

Sammendrag

Håndtering av brukerinput i web-applikasjoner har ved flere anledninger vist seg å være en vanskelig oppgave. Gang på gang har vi sett sårbarheter i slike applikasjoner på grunn av måten input benyttes uten å sikres først. På grunn av web-applikasjoners dynamiske natur er det også vanskelig å automatisere slike tester, da brukergrensesnittet er i stadig endring. En test-metode må derfor ta hensyn til dette. Videre er det urimelig å teste alle mulige verdier en bruker kan taste inn, og et subsett av mulige verdier må velges. Å overlate ansvaret for å finne dette utvalget til programmereren kan vise seg å være uklokt, ettersom han ofte kun tester de verdiene han vet er sannssynlige.

I denne oppgaven beskriver vi en metode og et verktøy for (semi)automatisk generering av tilsynelatende tilfeldig test data (også kjent som «fuzzing»). Testmetoden og verktøyet vårt har blitt anvendt på flere populære frie web-applikasjoner, og eksperimentet vårt viser at fra testerens perspektiv er metoden rask, enkel og effektiv. Ved å benytte metoden og verktøyet klarte vi å avdekke problemer i flere av applikasjonene vi testet.

En artikkel-versjon av oppgaven finnes i Appendix C.

Nøkkelord: D.2.5.k Testing strategies, D.2.5.o Test execution, D.2.5.r Testing tools.

Acknowledgements

Writing a master thesis is a lot of work, and even though there is only one author, there are people behind the author who deserves thanks. First of all, my supervisor, Einar Snekkenes, who has provided me with advice on how to conduct the experiment, how to find more related work and how to best lay out the thesis. Snekkenes was also the one who encouraged me to write the article version found in Appendix C and submit it to a conference.

Other people have also helped this thesis become what it is, and I'm even more grateful they chose to help me, as they don't necessarily have the background knowledge my supervisor has, but they took some time to help me in their own ways. Trond Viggo Håpnes provided me with books on software testing and what he calls "balcony beers" in sunny afternoons. Yngve Solberg helped me with some proof reading and gave some pointers on typographic mistakes and inconsistencies. I would also like to thank my opponents, Terje Risa and Tron Ingebrigtsen, for their valuable feedback on the thesis and extensive list of possible improvements.

Trine Sundstad deserves thanks for accepting my long hours at school, as well as helping to take my mind off the thesis in the afternoons. My father, Morten Hammersland, along with my mother, Inger-Elisabeth Hammersland, encouraged me to keep going when my motivation was low. My father also took some time to read through both the thesis and the article, and gave some good tips on what needed improving, as well as possible angles to difficult problems I faced. I appreciate the support greatly, and I'm looking forward to celebrating their silver wedding anniversary this summer.

Last, but not least, I'd like to thank my classmates for making our time in the master lab enjoyable — and caffeinated. Together we have solved problems with LaTeX, exchanged hints on experiments and layout of thesis and presentation, participated in each others experiments, and of course done some high quality procrastination involving baked potatoes, ice cream and various short snippets of video found on the Internet. I will no doubt remember this last semester, and the fun we had together on the lab, for a long time.

To everyone who has helped me with the process of writing this thesis: I am forever thankful.

Rune Hammersland, 2008-06-30

Contents

List of Tables

List of Figures

Code Examples

1 Introduction

This chapter starts off with an introduction to the topic covered by the thesis, and a description of the problem we are trying to solve. We will look into the motivation behind this work, and our research questions are listed. We will also describe the research method used, and give an overview of the contributions this thesis provides.

After this chapter, the thesis will follow this outline:

Chapter 2 Contains an introduction to software testing and related work on fuzz testing.

Chapter 3 Contains an introduction to how web applications are composited and how input flows from the user to the application and back again.

Chapter 4 Explains the method we use to fuzz test web applications, building upon previous approaches.

Chapter 5 Explains how we created a prototype for fuzzing web applications, based on the previous method.

Chapter 6 Gives details on how to use the fuzzer built in the previous chapter.

Chapter 7 Details surrounding the project we conducted to test the method: set up of environment, list of tested applications, and details about the outcome of the experiment.

Chapter 8 Lists the contributions made in this thesis.

Chapter 9 Provides a discussion on the findings.

Chapter 10 Contains a list of possible improvements, and suggestions for future research.

Chapter 11 Conclusions for this thesis.

1.1 Topic Covered by the Thesis

This thesis is about testing how web applications handle user input. We evaluate how well a method called “fuzzing” applies to this problem by outlining a way to apply this test technique to web applications, a way to implement this proposed method, and finally by an experiment against popular open source web applications, as well as a couple of new applications.

Fuzzing is a testing technique developed by Barton P. Miller at the University of Wisconsin in USA. As they state in their first paper on the subject [1], “it started on a dark and stormy night” when one of the authors experienced line noise on his connection to the university. They state that the “line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash.” Using the experience from this night, they created an experiment where they fed random input to various programs to see what happened. In later studies [2, 3, 4], they also tested GUI (Graphical User Interface) programs for several systems by sending random key presses and random mouse events.

Using this technique, they discovered that several programs didn't handle random input too well, many of them crashing. Where source code were available, they studied the "core dump"[1] and source code to find out where the problem occurred. Many of the problems were due to simple mistakes as neglecting to check the return value of functions before using the result. For a short introduction to fuzzing, you could read Sprundel's article from the 22nd Chaos Communication Congress [5].

Little or no research has been done on using fuzz testing to test web applications. There are some tools available: Paros[2], SPIKE[3] and RFuzz[4] to mention some. The first two work by acting as an HTTP proxy which allows you to modify POST or GET values passed to a web site. The last one is more like a framework for fuzzing which enables a programmer to programatically fuzz web sites and, optionally, generate statistics through the generated CSV files. We have also looked at The Burp Suite, Peach Fuzzing Platform and the Sulley Fuzzing Framework.

1.2 Keywords

These are the keywords covered by this thesis:

D.2.5.k Testing strategies.

D.2.5.o Test execution.

D.2.5.r Testing tools.

1.3 Problem Description

As evidenced by Miller et al., many applications are not robust enough against random input. While they have researched how fuzzing affects command line and GUI applications, little, or no research has been done on how it affects web applications. Tools do exist, but to the writer's knowledge, no reports have been published on how web applications stand against fuzzing. With the ubiquitous blogs and user contributed websites that exists in this Web 2.0 world, it would be interesting to find out how robust some of the popular applications are. When handling large amounts of user input, it is important that there is no way that input can put the web application in an undefined state, in other words: crashing it. Many programmers choose to use a web framework to avoid having to handle these problems themselves, and others make their own frameworks to simplify things. In both cases erroneous user input might affect their application, as nothing will prevent you from doing "stupid" things as evaluating the user input as code (e.g. if you're using the `eval` function in dynamic languages like Perl). Articles have been written on how a programmer can evaluate untrusted code "safely" (e.g. through sandboxing), however, that is outside the scope of this thesis.

Fuzzing has already proven to be successful for many fields. This thesis looks at how to implement a fuzzer suitable for fuzzing web applications, and how well this testing technique fits with web based applications.

[1] Most systems can be configured to leave a core dump when a program crash. The core dump contains information about what the program had loaded in memory and registers at the time of the crash.

[2] http://www.parosproxy.org/

[3] http://www.immunitysec.com/resources-freesoftware.shtml

[4] http://rfuzz.rubyforge.org/

1.4 Justification, Motivation and Benefits

Because so many web sites gives users the possibility to collaborate and contribute to the site, they are also vulnerable to erroneous input and / or users with bad intents. By typing in random data in the fields provided, either by accident, or by intent, the users may put the web application in an undefined state, where it will no longer respond to new requests. Using random testing, malicious users might also be able to discover other weaknesses in the application, like unsafe handling of input leading to a command injection vulnerability or a way to manipulate data stored in a database, or changing the pricing of an item they buy on an e-commerce site.

Through fuzz testing, we can find out how well the web applications handle random input, and not the input the programmer expected (whether legitimate or illegitimate input was expected). By discovering where the applications fail to handle the fuzz data (random input) in a controlled manner, we can find out which programming practices resulted in the bad code, and possibly correct the mistakes made.

A good reason for looking into fuzzing for web applications is that producing a simple web application has a relative low cost. Web programming is also associated with a learning curve that starts out low: beginning programming for the web is easy, as is creating minor programs, but the bigger the program, the harder it gets (especially with regard to security). As web programming is considered “cheap” and easily accessible, so should testing techniques. A company that invests a small amount of effort in creating a simple application (e.g. for internal use), shouldn’t need to invest a great amount of effort in testing it. Fuzz testing is considered easy to automate and easy to use, so it should fit the bill nicely.

1.5 Research Questions

The main questions we are looking to answer is: To what extent is fuzzing suitable for testing web applications? We will try to find this out by answering the following questions:

- How much work does it take to implement the fuzzer?
- How effective is it? To answer this, we will look at:
  - Automation — is it possible to automate, and to what extent?
  - Finding bugs — by setting up a test environment, will we be able to find bugs using this testing method?
- If we find bugs, what kind of bugs are they?

As stated earlier, research has been done on how well command line and GUI applications handle fuzz data. Some research has also been done on fuzzing for network protocols, but to our knowledge, similar tests have not been done on web applications.

1.6 Research Method

We started our work with looking for related work. The later papers by Miller et al. had some pointers to other places we could look. In addition they provided us with a good set of keywords to use when searching for articles. While looking for related work, we quickly found that some work on the area had been done, but mostly outside academia, so few papers were produced. What we found gave us a starting point for attacking our problem.

After reviewing related work, we used some of the ideas found in the studies by Miller et al. and refined them to fit better with testing web applications. The method we came up with for testing was then prototyped, and the resulting prototype was used in an experiment. Again the related work gave pointers on how this should be conducted. Most of the studies by Miller et al. has been quantitive studies, but they also contain a small element of a qualitative study. They achieve this by testing a large amount of programs (the quantitative part), and when the testing is done, they dive into some of the faults to discover what triggered them (the qualitative part). We felt that this was a good approach, but setting up an amount of web applications matching the amount of pre installed command line utilities tested by Miller et al. is very time consuming, so we settled for a smaller amount of test subjects.

1.7 Summary of Contributions

In this thesis we propose a method for creating a fuzzer suitable for fuzzing web applications. We have implemented a tool chain that uses this method, and have applied these tools to several popular web applications available for installation on a computer to see how they handle fuzz data as input (we have not been looking at how fuzz testing affects hosted solutions, such as YouTube, as testing other peoples production systems is considered unethical).

We present a listing of flaws found in the web applications tested in Section 7.3, and where possible we include information on why the application failed, and how to fix the mistake, similarly as what Miller et al. did in [4]. We also considered checking how these applications stand against SQL injection attacks and cross site scripting attacks, but we found that this was not directly related to the random testing technique we know as “fuzzing”, as more directed attacks, with a specific payload, would be necessary.

2 Related Work

This chapter gives a short introduction to software testing and an overview of the related work on fuzzing. Section 2.1 gives an example of how testing techniques can be organized in a hierarchy as well as an introduction to some common testing terms. Section 2.1.1 introduces a number of dynamic testing techniques, and tries to establish where fuzzing fits.

Section 2.2 introduces fuzzing, and Sections 2.2.1 through 2.2.5 explains how fuzzing has been used to discover weaknesses in command line applications, GUI applications, programming libraries and WiFi drivers respectively. At last, Section 2.2.6 lists a couple of existing tools for web application vulnerability discovery which includes a fuzzer.

2.1 Testing

Testing techniques can be divided in a hierarchy like shown in Figure 1. This makes it easier for us to define the scope of our thesis, by explaining where fuzz testing belongs, and which techniques are irrelevant for this kind of testing. While the terms in Figure 1 can apply both to hardware and software testing, our emphasis is on testing software.

The first division is done between static and dynamic testing. Static testing requires no execution of the software, and is done on the code base. According to Ryber [6] this is often done by hand, and techniques include inspection, walkthrough and different kinds of reviews. One form of review is when newly written code has to be approved by one or more coworkers before being applied to the code base. Compiling code might also be considered a static testing technique, as the compiler parses the code in order to make machine code. Compilers can point to errors by analyzing the code before it is being run.

Dynamic testing is when we are testing running software. This can be done in a number of ways, one of which is widely known under the term “debugging”. Debugging is done by attaching a program known as a “debugger” to the running software. Using the debugger, the programmer can halt execution of code, inspect the memory of the application, alter the program flow and step through the code one step at a time. Other dynamic methods include code coverage and unit testing.

We see from the figure that dynamic testing can be subdivided in two groups: black box testing and white box testing. Sometimes the terms “behaviour based testing” and “structural testing” are used, but we will use the former terms. Black box testing is when we consider the software as a black box: we know nothing of what it contains, but we know what input it can take, and we know what output we might expect. White box testing is the opposite: here we have access to the source code, and in addition to knowing the inputs and possible outputs, we also know how the “box” converts the input to the output. Debugging usually falls into the white box category, while unit testing can be placed under both, depending on how it is used.

At last, both black box testing and white box testing can be divided in “functional tests” and “non-functional tests”, where the latter also is called testing “quality requirements”. Functional

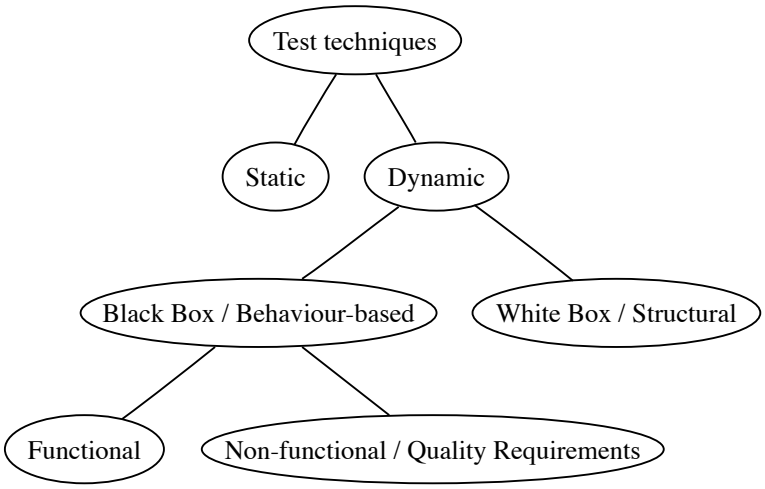

Figure 1: Division of testing techniques into a hierarchy. Figure taken from [6]

tests are tests which are aimed at testing specific functionality provided by the software, while non-functional tests are aimed at testing other qualities like the ones defined by ISO 9126 [7] (emphasized):

1. Functionality — *A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.* Ryber's interpretation: "are the desired functions present?"
2. Reliability — *A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.* Ryber's interpretation: "Is the system robust, and does it work in different situations?"
3. Usability — *A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.* Ryber's interpretation: "Is the system intuitive, comprehensible and simple to use?"
4. Efficiency — *A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.* Ryber's interpretation: "Does the system use resources well?"
5. Maintainability — *A set of attributes that bear on the effort needed to make specified modifications.* Ryber's interpretation: "Can the workforce, developers and users upgrade the system when needed?"
6. Portability — *A set of attributes that bear on the ability of software to be transferred from one environment to another.* Ryber's interpretation: "Can the system work on different platforms, with different databases, etc.?"

Ryber [6] notes that non-functional testing is often best to leave to the people with the domain knowledge. Usability testing should preferably be done by experts in usability and testing of

reliability and efficiency might be smart to leave to the developers, as they have the tools needed. We also note that efficiency testing is best done as white box testing, since this makes it easier to find the bottlenecks in the source code, and which functions that pays off to optimize.

2.1.1 Techniques Suitable for Dynamic Testing

In this section, we'll quickly describe some testing techniques that are relevant for dynamic testing. These techniques will work with both black box testing and white box testing.

Equivalence partitioning This method divides the possible inputs into groups, and works under the assumption that testing two kinds of input from the same group yields similar results. I.e. by defining a group of allowed input values and a group of disallowed values, trying two inputs from the group of allowed values should both yield a success, and consequently, trying two values from the group of disallowed values should yield a failure in both cases. Note that there might be several groups of disallowed input defined as two disjoint sets. An example could be a grading scale $\{A, B, \ldots, F\}$. Allowed values would be the given set, and examples of disallowed values could be: $i \in \mathbb{N}$, $j \in \mathbb{R}$, $\{G, H, \ldots, Z\}$ and an empty string.

Boundary value analysis This method looks at the allowed input values, and tries the values found on the boundaries between allowed and disallowed values. If a function for reserving tickets to a movie has an allowed maximum of 6 tickets, and a minimum of 1, we see that the range $\{1, 2, \ldots, 6\}$ is the allowed values, and all other integers are disallowed. So-called "off-by-one errors" are common in programming, and boundary value analysis tests these cases. The boundaries in the function mentioned will be values around the start and end of the allowed range, i.e. $0, 1, 6, 7$. In these cases, we will often test negative values as well as real numbers. This method is most used in white box testing — with access to the source code, we know which values are allowed and which are not. It applies to black box testing when we can derive the allowed values, or when they are stated in the specification or somewhere else.

Combinatorial analysis This method works best if there is a limitation on the possible inputs, and is based on trying all possible inputs. If a function has several inputs, this can quickly lead to what is called a "combinatorial explosion." The testing method can however be used on functions with a small input space, and is possibly best suited for black box testing of a function where the input limitations are not clear.

Experience based testing People experienced in testing software systems might lean on their experience in identifying possible weaknesses. By having tested similar functions or software earlier, they might know which values and boundary cases that are likely to yield an error. Reliance on earlier experience is also a part of "exploratory testing" [8, 9].

Random testing This method, briefly mentioned as "ape testing" on page 87 in [6], is based on sending random input to the software or function, in the hope of discovering errors. The technique doesn't seem to make much sense for white box testing, and is most commonly used under black box testing. It can, however, be a quick and easy supplement to other white box testing techniques. This method is the basis for our thesis.

2.2 Fuzzing

> "Our testing, called fuzz testing, uses simple black-box random input; no knowledge of the application is used in generating the random input." — Forrester and Miller [3]

As Miller et al. [1, 2, 4] and Forrester and Miller [3] already have stated, many applications are vulnerable to buffer overflows and similar attacks because of bad programming practices. Many of these flaws are hard for the programmer to spot, as they often make the assumption that a function cannot fail and hence they do not check the returned value. Fuzzers can assist in these cases, as backed up by Oehlert [10], a software engineer at Microsoft who found several flaws in Microsoft's HyperTerm after using a fuzzer to provide semi-valid input to the program. Microsoft's "Trustworthy Computing Security Development Lifecycle" [11] even states that "heavy emphasis on fuzz testing is a relatively recent addition to the SDL [Security Development Lifecycle], but results to date are very encouraging."

While many papers have been written on fuzzing, they have mainly focused on client software on the computer, and in some cases, like Xiao et al. [12], on network protocols. What seems to be missing is research on how web applications can be tested randomly using fuzzing, and which flaws might appear. Several papers, like [13], have suggested that user input is a huge problem for web based applications, and especially regarding command injection attacks. Many injection attacks are based on buffer overflows, which fuzz testing seems to be good at discovering.

Enumeration attacks (similar to combinatorial analysis) might be a better approach for discovering vulnerabilities in web applications, but should not be confused with fuzzing. While Dafydd Stuttard and Marcus Pinto writes about fuzzers in their book about "hacking" web applications [14], they seem to mistake the primary idea behind a fuzzer — at least according to the definition by Miller. They mainly use the fuzzer as a means for enumerating attacks. A true fuzzer should try strictly random input, or a combination of valid and random input. If you are sending input based on a list of "possibly malicious input" or based on incrementing values, you are doing an enumeration attack and not a fuzz attack. Stuttard and Pinto also states that analyzing results from web application vulnerability discovery is hard, and manual work is often required. Using the HTTP status code is one way of automating this task, but it might not be clear if an error in the status code indicates an actual error — and on the other hand, an application doesn't need to fail in order to be vulnerable (i.e. cross site scripting attacks, path traversal attacks and command injection attacks).

The rest of this chapter looks at different targets for fuzz testing, and what research has been done on the subject. Bear in mind that while fuzzing has existed since the early nineties, it is still a rather new tool and has only lately been getting more attention. Many of the people who employ fuzzing as a means of security testing are not working in the academic field, so some of the references are bound to be of a somewhat lower quality than usually expected from a research standpoint. This certainly doesn't mean they are bad articles (e.g. the articles by Miller et al. [15, 16] are well written and contains lots of technical details). It only means that some of them are not published in academic publications and may not follow the same style of writing which seems to be expected from academic work.

Client running under:	[1]	[2]	[3]	[4]	[17]	[18]
UNIX CLI	24-33%	43%			11-29%	
GNU/Linux CLI		9%			4% ,9%	
X11		26%, 58%				
Windows GUI			45-97%			
Windows CLI						23%
Mac OS X CLI				7%		
Mac OS X GUI				73%		

Table 1: Overview of results from studies on client software. Where only one number (or a range) is listed, this indicates how many programs crashed or hung. Where two numbers are separated by a comma, the first number represents crashes and the second hangs.

2.2.1 Command Line Applications

In [1], Miller et al. tested command line programs on seven different versions of UNIX (between 49 and 85 programs, depending on the system, usually between 70 and 80), and managed to make 24-33% of the programs hang or crash (depending on which version of UNIX they tested). The lowest (24.5%) was on an IBM machine running AIX 1.1, and the highest (33.3%) on a HP machine running 4.3BSD. When they redid the study in 1995 [2], only 9% of the programs crashed or hung on the Linux machine (running Slackware 2.1.0), and on the other end 43% of the programs had problems on the NeXT machine (running NEXTSTEP 3.2). In this study between 47 and 80 programs were tested on each platform (usually between 70 and 80). Results on fuzz testing X applications (38 applications) were published in the same study, showing that 26% of the X applications crashed when tested with random legal input (events), and 58% crashed when given totally random input (events).

Bowers, Lie and Smethells redid the studies Miller et al. did on UNIX command line programs in their study from 2001 [17]. To accommodate for the fact that some of the programs originally tested had since become abandoned, they changed some of the programs for newer alternatives, (e.g. replacing vim for vi). Bowers et al. also noted that the fuzz program Miller et al. created itself contained a bug that might have been found using fuzz testing. This shows that these kinds of mistakes are very easy to make, even though you are aware of the problem. The study did by Bowers et al. shows that the open source community had taken notice of Miller's study [19], and had improved the stability of many of the affected programs. Bowers et al. also notes that many UNIXes have added warnings to the man-pages of dangerous functions after the Miller study, an example being `man 3 gets` on e.g. GNU/Linux and Mac OS X:

```
BUGS
       Never  use  gets().  Because it is impossible to tell without knowing
       the data in advance how many characters gets() will read, and  because
       gets()  will  continue to store characters past the end of the buffer,
       it is extremely dangerous to use.  It has been used to break  computer
       security.  Use fgets() instead.
```

2.2.2 GUI Applications

In Forrester and Miller's study on Windows [3] 33 GUI programs were tested on Windows NT 4.0, and 14 GUI programs were tested on Windows 2000. In this study three methods were used in order to send input (events) to the applications: the `SendMessage` and `PostMessage` API calls, as well as "random valid events". The study showed that using the API to send random messages to the running programs caused more errors than sending valid random events. Using `SendMessage` they achieved an error rate of 81.7% on NT 4.0 and 85.7% on 2000. Using `PostMessage` the error rates were 96.9% and 71.4%, and using "random valid event" they got 45.4% and 64.3% error rates. They explain that the API calls contains messages with pointers as parameters:

> "[...] which the applications apparently de-reference blindly."i — Forrester and Miller [3]

The "random valid events" are also a better measure of the reliability, as the messages sent through `SendMessage` and `PostMessage` usually comes from the kernel. Ghosh et al. also looked at the robustness of Windows NT software using fuzzing [18]. They only tested 8 different programs, but had a lot of different test cases where they found that 23.51% of the tests resulted in a program exiting abnormally and 1.55% of the tests resulted in a program hanging.

The last study from Miller et al., conducted on Mac OS X [4], shows similar results to the best results from [2] when it comes to command line programs. In this study, 135 command line programs were tested (over 1.5 times as many). The results: only 7% of the programs crashed or hung. In other words this means that command line programs have become very good at handling bad user input. This comes as no surprise, as most of the command line programs in Mac OS X are GNU programs[1], or programs who also have been developed in an open source fashion. There are only a few programs (10 of 135) that Apple has created themselves. The GUI applications on Mac OS X had a worse fate. Of 30 tested programs, 22 crashed or hung, yielding a 73% failure rate; the worst in our overview (Table 1).

A similar technique to fuzzing was used during the development of the Macintosh 128k which was released in 1984, but looking at the results from Miller's study, similar tools are probably not used today. The developer team on the Macintosh 128k created a program they called "The Monkey" [20] which used some APIs to send random events to the operating system

> "[...] so the Macintosh seemed to be operated by an incredibly fast, somewhat angry monkey, banging away at the mouse and keyboard, generating clicks and drags at random positions with wild abandon." — Hertzfeld [20]

This program was a great help in the quest for bugs. Similarly there exists a program for modern UNIXes called `crashme` which has been of great help for developers of GNU/Linux in identifying rare cases where the system would crash due to erroneous input. [21] implies that many commercial UNIX versions fail quickly after starting `crashme`, while GNU/Linux are resilient against the kind of attacks used. In a whitepaper submitted to the "Black Hat USA 2007 Briefings and Training" conference [16], Miller and Honoroff outlines several useful utilities and tips for fuzzing software on Mac OS X. They also note that OS X comes with several useful tools to aid this kind of testing.

[1] See e.g. `bash`, `grep` or `gzip` at http://directory.fsf.org/GNU/

2.2.3 Programming Libraries

Random testing has also been applied to programming libraries. Claessen and Hughes have developed a tool for the Haskell language called QuickCheck [22] which uses a specification for how a function works (called properties of the function), and proceeds to test the function with a large number of automatically generated test cases. A property can be that a list xs, reversed two times, should equal the original list of xs. The property also states the type used, for example each $x \in xs$ is an integer. QuickCheck will then test this property for a long range of random lists of integers to see if the property holds. Similar tools to QuickCheck exists for other programming languages like Common Lisp, Erlang, ML, Python, Ruby and Scheme.

In [23], Kropp et al. conducted a fuzz experiment on a long range of POSIX functions by abstracting different datatypes. Using that approach they could generate semi-valid input for the functions and test how well they handled random integers, strings without the termination character, open filehandles and more. That paper also showed that using a good enough fuzzer, you can expect about the same results as when you are writing exhaustive tests. Since the random number generator was seeded with the name of the function they were testing, the same random sequence was used every time they tested the same function.

Schmid and Hill have also looked at semi-valid input versus random input for testing of API functions [24]. While Kropp et al. looked at POSIX functions, Schmid and Hill looked at the WIN32 API functions and some command line tools. This study shows, contrary to Kropp et al., that semi-valid input yielded more failures, and hence was a better way to test.

2.2.4 Network Protocols and the Web

Banks et al. [25] points out that while many fuzzers exists for fuzzing network traffic, like SPIKE [26] and PROTOS [27], they don't handle stateful protocols very well, and making them do so might require more work than writing a new framework altogether. Their creation — SNOOZE — parses XML documents containing possible states of an application (represented as nodes in a graph), and the available transitions between the states (represented as edges in the graph), along with an XML document explaining the possible messages, along with default values. Using these components they can write a script that creates fuzz values for some of the messages, and thus they can control which point in the protocol state machine they wish to attack. If they had been using a more standard fuzzing approach, they would rarely have gotten past the initial state, but using their own approach they were able to find bugs in SIP (Session Initiation Protocol) implementations that were "hidden deep in the implementation of [the] stateful protocol."

Fuzzing has also proven effective in discovering vulnerabilities in web browsers, and through this a means of exploiting the Apple iPhone [15]. As the authors of the whitepaper state: "Such fuzzing can be performed against applications such as MobileSafari or against the WiFi or BlueTooth stack. The vulnerability we discovered and exploited was found in MobileSafari using fuzzing." The infamous "Month of Browser Bugs" article series (which has since been removed) also utilized fuzz testing in order to discover vulnerabilities in the most commonly used web browsers [28]. The project was criticized for what some saw as an "irresponsible disclosure of

vulnerabilities"[2], but the author (who is also the author of the Metasploit project) insists that all vendors were made aware of the vulnerabilities before he disclosed them[3].

2.2.5 Wireless Drivers

Many wireless drivers are unfortunately closed source, as the vendors creating the cards believes they are letting go of their "intellectual property" by disclosing the source code. The result of this is that the only way for "ordinary people" to test these drivers is through black box testing (or indeed reverse engineering — discovering the workings of a device through analysis — which has a lot in common with black box testing). This is especially true for closed devices like PDAs. Testing of wireless drivers is very interesting these days, as wireless connectivity is becoming the standard for many people. It is made even more interesting by the fact that wireless drivers runs in kernel mode (at least on operating systems in common use), and thus an exploit can get full access to the computer, with the attacker only in proximity of the victim. Butti and Tinnès stresses this fact in their paper on discovering and exploiting wireless drivers [29], as well as the fact that the wireless networks are weakening the security perimeter.

Mendonça and Neves has done some preliminary testing of the wireless drivers in an HP iPAQ running the Windows Mobile operating system [30]. Without having the source code available, they started writing a fuzzing framework targeting the wireless drivers on the device. By running a monitor program on the device they have been able to find some weaknesses while fuzz testing the driver. Whether the weaknesses are exploitable had not been discovered by the time the article was published. However, Butti and Tinnès were successful in discovering and exploiting the madwifi driver running in the GNU/Linux kernel, as well as finding several denial of service vulnerabilities in different wireless access points. Some of the findings were included in the Month of Kernel Bugs (`http://projects.info-pull.com/mokb/`) project and included as modules in the Metasploit project (`http://metasploit.com/`). The exploit targeted at the madwifi driver is a remote exploit giving the attacker a shell with the privileges of the user performing a scan of available access points (root or normal user through the `iwlist` command, or the user running `wpa_supplicant` — usually root). The article contains information on how they managed to exploit the driver, and how they made sure that the wireless stack would still be functioning after the attack.

Program	Target	URL
AppScan	Web apps	`http://www.watchfire.com/products/appscan/`
Burp Suite	Protocols / Web apps	`http://portswigger.net/suite/`
Peach	Protocols	`http://peachfuzz.sourceforge.net/`
Sulley	Protocols	`http://fuzzing.org/sulley`
WebScarab	Web apps	`http://www.owasp.org/`

Table 2: Different fuzzing tools

[2] See e.g. `http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9001610`

[3] `http://blog.metasploit.com/2006/07/month-of-browser-bugs.html`

2.2.6 Existing Tools Suitable for Fuzzing Web Applications

We have looked at existing tools for fuzzing web applications. Some of the applications we found were mainly focused on protocol level fuzzing, but there exists some multipurpose fuzzers which also aims at finding vulnerabilities in web applications. The fuzzers we looked at were Burp Suite[4], the Peach Fuzzing Platform[5], the Sulley Fuzzing Framework[6] and WebScarab[7].

The Burp Suite (introduced in the book by Stuttard and Pinto [14]) implements a web proxy which you can configure your browser to use. Every request made from the browser will then be displayed in an application. From there you can pick the requests you want to use as entry points for fuzzing or other attacks. The Burp Suite implements a bigger range of attacks, and is not strictly a fuzzing tool. Due to this, the interface is rather crowded with buttons and tabs, so getting started might not be easy. One of the most useful components, the Burp Intruder, is only distributed in a limited version unless you pay for the program.

Peach also looked promising, and is being developed in an open source fashion. This framework emphasize reusability, and writing components for use in one project will likely yield useful components for other projects. It is mentioned in the book by Sutton et al. [31] as having a rather steep learning curve, and the author of the framework seems to agree with this. Lately the focus has been on improving the ease of use and making the framework available to people who doesn't know how to write code. This framework focuses on fuzzing network protocols, and building a custom fuzzer for one web application would amount to a great deal of work. The realistic scenario would be to first implement components to ease the task of creating fuzzers for web applications.

The Sulley Fuzzing Framework [31] also seems to be aimed primarily at fuzzing network protocols and could be suitable for fuzzing web applications by targeting the protocol layer. However, in our short review of the feasibility of this framework, it seemed like it depended heavily on process monitoring in order to achieve great results. The example given in the book (and documentation) shows that a process monitor is reporting information about the target of the attack, and is able to restart it if it has failed. This seems great if we were fuzzing applications with a web interface, and not web applications invoked by a web server like e.g. Apache.

WebScarab uses the same approach as the Burp Suite. You connect your browser to a custom proxy which intercepts requests and responses. As with Burp Suite, WebScarab implements several attack techniques, and amongst them — a fuzzer. When inspecting the fuzzer, we found that you need to provide it with sources for data. Sources can be defined as a file containing one entry per line (e.g. the system dictionary file), or as a regular expression. When supplying a file, the "fuzzer" seems to iterate through the file, until all the entries are used. To get better randomness, we tried supplying the random device (`/dev/random`) as the file, but the program tried to read in the "entire" file in order to display the values. This caused the program to raise an exception, and the device could not be used. As the fuzzer in WebScarab is not working more like an enumerator than a generator of true random input, we discarded it.

[4] http://portswigger.net/suite/
[5] http://peachfuzz.sourceforge.net/
[6] http://fuzzing.org/sulley
[7] http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

Late in the process of writing this thesis, we received a tip for a product named AppScan[8]. This is a commercial vulnerability discovery tool created by the IBM-owned company Watchfire. The product pages mentions that fuzzing is possible through the "pyscan" extension, but it seems like the main method used is scanning pages for known vulnerabilities. However, since the price of the product is rather steep[9], we decided to not evaluate this program. We could have gone for the trial version, but this is restricted to testing a predefined website which is outside our control.

[8] http://www.watchfire.com/products/appscan/

[9] IBM's pages states USD $4000 for "Enterprise Edition" and USD $6000 for "Tester Edition." See e.g. https://www-112.ibm.com/software/howtobuy/buyingtools/paexpress/Express?part_number=D61V2LL,D61V3LL,D61UYLL,D61V0LL,&country=USA

3 The Anatomy of a Web Application

This chapter will give a brief overview of what a web application is, and how it works. Section 3.1 gives an introduction to how pages are identified and how they are linked together. Section 3.2 explains how a browser works to get a page from a web server, by giving a short introduction to HTTP and HTML. Section 3.3 explains how a user can give input to a web application, and how the result is communicated back to the user. Section 3.4 will go a bit deeper into the concept of HTTP status codes, as these will be important to us when analyzing the results later.

3.1 A Collection of Pages

One way to look at the application is as a collection of webpages that work together to achieve the goal of the application. Each page is identified by a unique URL (Uniform Resource Locator) which acts as a pointer to the page. By separating the URLs by the forward slash, we see that the pages of many applications are located in a tree structure which is reflected in the URL. We have given an example in Figure 2. In this figure, the red edges labelled “auth” represents a subtree that is restricted to authorized users. Historically the URL gave a pointer to where in the file system the page was located, but with a technique known as URL rewriting, this is not necessarily the case anymore.

The pages can contain links to other pages in the application. If we regard the pages as nodes and the links as edges, we can create a graph illustrating the possible connections between pages in the web application, as seen in Figure 3. When we later talk about crawling a page, the graph model is a convenient way to visualize how the crawler must work and which constraints it can bump into. Looking at the figure, we see that the root of the web application has a link to what seems like an administrative interface. Since we need to be logged in to access the page, a crawler might be able to see that the page is there (through the link from the application root), but it might not be able to access the page, and other pages that might be hidden behind it. The dashed edges indicates links that exists but is impossible to find unless logged in; similarly the dashed node indicates a page that unauthorized users have no knowledge of.

3.2 Getting a Page

By issuing a HTTP GET request on one of the URLs, your browser will ask the web server to retrieve the page associated with the URL. The web server will respond by sending an HTTP response with some HTTP headers, and a message body, as specified by the HTTP 1.1 RFC [32]. For a web application, most message bodies will be formatted in the hypertext markup language (HTML [33]), and this will be reflected in a HTTP header called “Content-Type”. A simple example of the HTTP request/response model can be given as follows:

```
GET /webapp/post/1 HTTP/1.1
User-Agent: HTTPClient/1.0
```

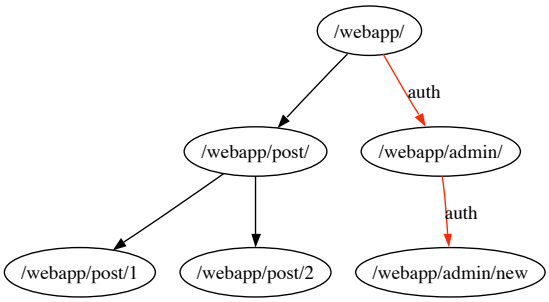

Figure 2: The hierarchy of a website as defined by the different URLs.

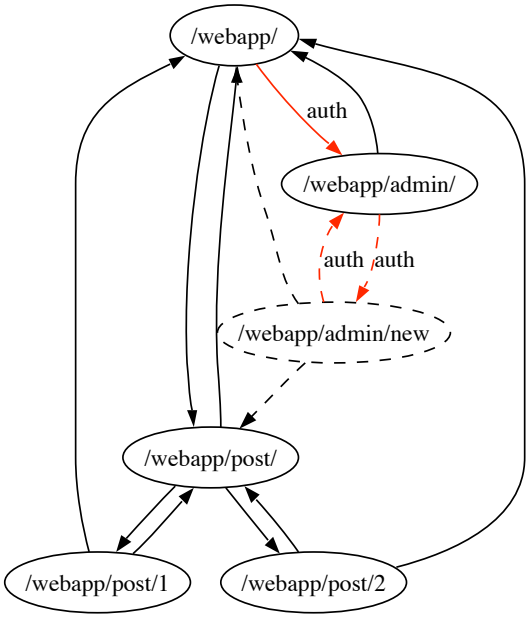

Figure 3: The resulting graph by crawling a website: the nodes are webpages and the edges are links between webpages.

Here, the browser has requested the resource associated by `/webapp/post/1` from the web server. A response could be received as follows:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 187
Date: Fri, 3 Apr 2008 23:59:59 GMT

<html>
  <head>
    <title>Webapp - Post #1</title>
  </head>
  <body>
    <h1>Some spectacular title!</h1>

    <p>This post will contain sensational news ... soon</p>
  </body>
</html>
```

We see that the headers and the message body is separated by a double newline and that the Content-Type header correctly specifies that the message body is of the mime type text/html. The

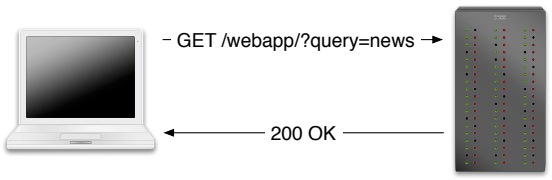

Figure 4: A simple HTTP request/response: A user has entered the value “news” in the field named “query” in a form that uses GET as the method. The application handles it without an error, and returns the status code `200 OK` (as well as more headers and a response body).

browser will then use it’s rendering engine to render the HTML document into what we usually associate with a web page.

3.3 Sending Input to the Application

The page might contain HTML markup for a form allowing the user to input some values. The form contains information on how to submit the contents — which HTTP method to use, and which URL the input should be sent to. By submitting the contents of that form, the user provides input to the application. The page identified by the form as the receiver of the input will usually use some server side script/program to handle the input, and display the result (or redirect to a page which does). Server side programming languages used in our experiment include PHP, Perl and Ruby. As an example, consider the following HTML form:

```
<form action="/webapp/" method="get">
  <input type="text" name="query" />
  <input type="submit" />
</form>
```

This tells us that the contents of this form should be sent using the HTTP GET method to the URL `/webapp/`. There are one field the user can fill in: a textbox for a query. There will usually be some explanatory text around the field as well, but this is omitted here. There is also a button to submit the contents of the form. Filling in the word “news” in the textbox and clicking the submit button will cause the browser to issue the request illustrated in Figure 4. In the figure, the response from the server indicates that the request went well, and the message body will likely contain some dynamic part which is dependent on the value submitted for the query field.

3.4 HTTP Status Codes

By looking at Section 10.4 of RFC 2616 [32], we see that the status codes in the 400 range are reserved for client errors which indicate that the fault is that of the client (usually the user or browser). Its Section 10.5 tells us that status codes in the 500 range are reserved for server errors, and “indicate cases in which the server is aware that it has erred or is incapable of performing the request.” Looking at other sections we can also see that a status code 200 means success and that status codes in the 300 range is used for redirection.

The information given to us in the status code is useful for many purposes. A web browser

can use it to transparently redirect the user to a new location if a web page has been moved (provided the old location has a redirect to the new), or to ask the user for credentials if the status code indicates that authentication is needed. An RSS reader will typically send a HEAD request to a page including a header called "Modified-Since" with the date it last checked for news. If the response has the status 304, it means the resource has not been modified since the last visit, and the reader does not need to issue a full GET request (easing the traffic load on the server). A status of 503 will tell us that the server understands the request, but is unable to give a response at this time (e.g. because of temporary overload).

The most important ranges for us will be the 200-, 400- and 500 range. This should tell us if our tests resulted in a success or failure, and if the server considered the failure to be on the client side or on the server side.

3.4.1 A Side Note on Redirection

Frameworks for writing applications for the web that supports the REST [34] philosophy that each resource should have a unique URL often uses redirection in the following way:

1. A user enters data into a form on the page with the intent of creating a new resource.
2. By clicking the submit button, the form data is sent to another location on the server using a POST method.
3. After handling the request (by e.g. inserting the data into a database), the application issues a redirect to a newly created location on the server, where the created resource is made available.

An example could be a user adding a new post to a weblog. First he fills in the fields `title` and `body` with the values "Title of Blog-post" and "This is the body of the post" respectively. Lastly he clicks submit to create the post. The browser then issues a POST request:

```
POST /webapp/admin/new/ HTTP/1.1
User-Agent: HTTPClient/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 58

title=Title+of+Blog-post&body=This+is+the+body+of+the+post
```

After creating the post (e.g. by inserting the values of title and body to a database), the application makes it available through another URL, and uses HTTP redirection to direct the user to the post:

```
HTTP/1.1 302 Moved Temporarily
Location: /webapp/post/3
Content-Type: text/html
Content-Length: 54
Date: Fri, 3 Apr 2008 23:59:59 GMT

<a href="/webapp/post/3">You are being redirected</a>.
```

At last, the user's browser issues a GET request to this new location:

```
GET /app/post/3 HTTP/1.1
User-Agent: HTTPClient/1.0
```

The application will then return the page containing the post created by the user. Typically, the information will be laid out according to a template. The application can also contain logic that e.g. presents the reader with links to other posts with similar content or a list of the most requested posts.

4 Method for Fuzzing Web Applications

The method we are using for fuzzing web applications resembles the one described in [1, 2, 3, 4, 31]. First we identify a number of web applications to test using fuzz testing (step 1 in Figure 5). The list of applications we have tested are given in Section 7.2.

User input (step 2 in Figure 5) in these web apps are mostly form based: a user fills in input elements in an HTML form and uses some method to send the input using (mainly) the HTTP GET or POST requests. We say "some method" because this might vary. Most of the times the user will submit the form using a submit button, but sometimes the page includes some JavaScript that sends the form contents on certain events. We will mainly focus on form based input through submits (no AJAXy sending). Testing AJAX components could also be done in a similar way, as these also rely on the web browser to send (mainly) HTTP GET and POST requests, and returns a regular HTTP response (with response codes and everything) containing either plain text or some XML.

Having identified how to send input to the applications, we can begin building a fuzzer that makes random input (step 3 in Figure 5) and, in turn, sends that input to the application (step 4 in Figure 5). How this was done in this project will be described in greater detail in Chapter 5. After building the fuzzer, we generate attack scripts which tells the fuzzer which host, port and paths to attack, and additionally which HTTP method should be used, along with possible query arguments, headers and cookies. The attack scripts we have developed allows the attacker to define global options like the hostname and port, but also cookies and headers that should be sent each time. Further it lets the attacker define a request that should precede each attack (say, if you need to log in before accessing a form). Finally the attacker can define attack points (paths with query options). Each attack can be launched either once, or many times, many being a configurable amount of repetitions.

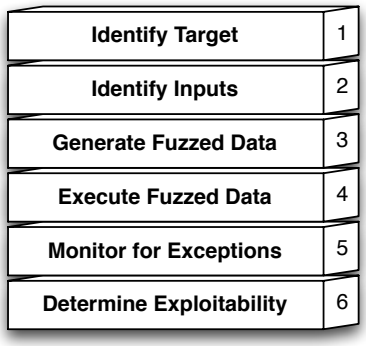

Figure 5: Different fuzzing phases, as defined by Sutton et al. in [31].

When an attack script is run, it logs all requests it sends, before awaiting the response from the web server. According to the HTTP 1.1 RFC [32], Section 6, the web server should supply a status line, along with headers and the message body. The different status codes used in the status line is given in Section 6.1.1 of the RFC, and explained further in Section 10 of the RFC. Having received the response, we log the entire response data for later analysis (a modified version of step 5 in Figure 5). We also log some statistics about how long the requests took, which are then totalled, in order to give us the sum, squared sum, number of requests, mean time per request, standard deviation, minimum and maximum request time. Using the logged responses we are also able to determine how many of the requests were handled correctly (step 6 in Figure 5).

By using a pseudorandom number generator to provide fuzz data, the scripts are possible to replay to achieve the same results. Details of the experiment are given in Chapter 7, and details surrounding the findings are given in Section 7.3.

5 Building a Fuzzer

In this chapter we propose a method to build a fuzzer suitable for fuzzing web applications. This method is based on the RFuzz library for the Ruby programming language, but we will point out where we are using existing code and how it can be implemented from scratch. An overview of how the parts are interconnected is presented in Figure 6.

For those who have never heard of the Ruby programming language[1], it is a multi paradigm scripting language with dynamic typing, not unlike Python. Ruby originates from Japan, and while everything in Ruby is an object, writing object oriented code is optional. The programmer is free to write procedural or functional programs, as well as object oriented ones. The language is inspired by features from languages like Perl, Smalltalk [35], Eiffel, Ada, and Lisp [36]. The main reason for using Ruby is the author's preference.

5.1 Creating Attack Scripts for Webapp Fuzzing

In order to specify how the applications should be attacked, we have created a way of writing attack scripts for fuzzing web applications. These attack scripts are rather standard Ruby scripts in which we can use some convenience methods for setting up a standard hostname and port for the web application, along with standard options like cookies and headers. Further down we specify "attack points" at the target site. These are mainly different web pages containing forms for user input. In the attack points the script writer can specify which path should be attacked and which method should be used (the standard HTTP methods are `GET`, `POST`, `PUT`, `DELETE` and `HEAD`, but we mainly use the two first) and which query options should be sent. See Listing 1 for an example of how this looks. In the listing, the variables `word` and `fix` are objects that yields different values each time their `to_s` (to string) method is called, the method `str(number)` yields a similar object, but bounded by the number. The `word` token will give different words, the `fix` token will give different "Fixnum"s (a 30-bit signed integer), and `str(50)` gives different strings with a length of 50 characters.

When the fuzzer is fed this script, the body is evaluated inside a new Target object. The Target object first sets up some default values for some of the variables, like "localhost" for hostname, and 80 for port. When the attack script sets a value for `@host`, it overrides the default value set by the initialization of the Target object (`@host` references an instance variable in the object). The `attack` method is defined to take a name and a block of code as a parameter. The code block is evaluated, and calls to `once` results in the following request getting queued once in the request queue. Calls to `many` results in the following request getting queued `@repetitions` times in the request queue. The number of repetitions is initially set to 50, but can be changed through the script. There is also a method called `before`, which takes a block of code (not unlike the `attack` method). If this method is used in the attack script, the attack method will add prepend

[1] http://ruby-lang.org

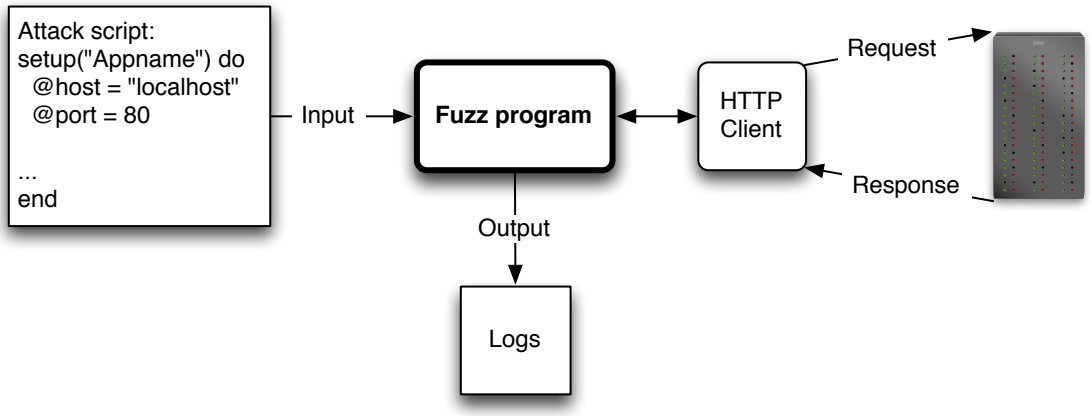

Figure 6: An overview of the main components in the fuzzer and how they interact. An attack script semi-generated by a crawler is fed to the fuzzer which in turn translates the attacks to HTTP requests which is sent to the target of the attack. The requests and their responses are then logged for manual inspection.

the before-block to each attack block. Thus providing functionality like the "fixture setup" found in unit tests. Using the before block, you can ensure that some requests will be made before each attack block, like say, logging in to the admin interface, or similar.

Creating these attack scripts by hand is easy, but tedious work. In order to automate this boring task, we also created small scripts for scraping web pages and generating an attack script. In their work on a vulnerability scanner looking for SQL-injection and XSS attacks, Huang et al. created a crawler based on Internet Explorer and a DOM (Document Object Model) parser to identify forms on webpages [37]. We use a simpler approach: the Hawler library[2] for Ruby combined with the Hpricot library[3]. Hawler is a simple web crawler, which scrapes all links on a page, and then does a breadth-first traversal. Every time it reads a page, it sends the URL, referrer URL and HTTP Response to a callback method. HPricot is a HTML parser which lets you traverse the DOM tree using XPath or CSS selectors.

We made two form-scraping scripts: The first script takes a complete URL as an argument, reads the web page it identifies and uses HPricot to find the forms, and filter out the interesting fields. Using the input fields, and attributes from the form, we can create the attack part of the attack script. See Listing 2 for an example of how we can scrape values from a form. The first script only parses one page. The second script reuses much of the same code, but as a callback method for the Hawler library. Each page the crawler reads is parsed by the callback in order to filter out forms. When the crawler starts, it outputs some generic information about the target (the `setup`-part), and the callback generates `attack` blocks. Finally, when the whole site is traversed, it outputs the end section. By calling this script and redirecting the output to a file, you get a good start for writing an attack script.

We did have some problems with the crawler. While you can pass headers which it uses in each

[2] http://spoofed.org/files/hawler/
[3] http://code.whytheluckystiff.net/hpricot/

```
setup "Webapp" do
    @host = "localhost"
    @port = 80
    @headers = "HTTP_ACCEPT_CHARSET" => "utf-8,*"

    attack "login-page" do
        many :get, "/login.php",
                   :query => {:user => word, :pass => word}
        many :get, "/login.php",
                   :cookies => {"administrator" => "true",
                                "username" => "admin"},
                   :query => {:user => fix, :pass => fix}
    end

    attack "post-page" do
        once :get, "/login.php",
             :query => {:user => :admin, :pass => :admin}
        many :post, "/post.php",
             :query => {:title => str(50), :body => base64}
    end
end
```

Listing 1: An example of an attack script

```
# form holds a form in the document.
# uri holds the uri of the document.
# if omitted, get is the default method
# if omitted, the path is this document.
method = form['method'] || "get"
action = form['action'] || uri.path

# filter input fields and return an array with values:
query = input(form).merge(select(form)).merge(textarea(form))
return [method.downcase, action, query]

# input method added for the example's sake
def input(form)
  # iterate over all input and button elements with a hash
  (form / 'input|button').inject({}) do |hash,item|
    # update the hash with this item's name as key,
    # and this item's value as value
    hash.merge({item['name'] => item['value']})
  end
end
```

Listing 2: Scraping a form

request, it is not straight forward to define pages it should abandon. This results in a problem when you add a cookie to the headers in order to "log in" to the admin panel and scrape these pages. The first couple of pages are usually parsed OK, but when it reaches the link that logs out of the admin panel, the rest of the URLs pointing within that password protected space will no longer be available. Since Ruby let you modify code dynamic, we could modify a method of the crawler to help prevent this effect. However we felt we got enough forms to test before it logged out. This should be dealt with later though.

5.2 Random Number Generator

We are using an ArcFour (RC4) random number generator which is provided by the RFuzz library. It is implemented as a C extension to Ruby, so it is pretty fast, and yields the same sequence of output given the same seed, making replay attacks possible. ArcFour is described in greater detail in [38] which also includes an example implementation in C. RC4 is usually used as a stream cipher by seeding the number generator with a secret key and XORing the plaintext with the output from the number generator, but we are using it as a "normal" random generator. As a stream cipher RC4 has proven to have weaknesses. Fluhrer et al. lists several weaknesses in the key scheduling algorithm [39], and how these affects protocols like WEP. Harris proposed methods of using the cipher in SSH (secure shell) that mitigates the weaknesses in the key scheduling algorithm to an acceptable level [40].

Utilizing the random number generator, we have provided convenience objects for usage in the attack scripts. By building a set of FuzzTokens, we provide building blocks to the script writer. Each FuzzToken subclass implements primarily one method, called `fuzz`. In this method it uses the random number generator supplied through it's superclass to generate random entities. The superclass also implements a `to_s` method which calls the `fuzz` method and returns the string representation of the result. The Target class supplies the attack script with a range of FuzzTokens, and as the `fuzz` method is only really called through the `to_s` method, they are evaluated every time the HTTP client creates a request (as the request path and parameters ultimately needs to be in string format).

We have not provided a method to fuzz file uploads, as few of the web applications tested do anything significant (like processing them) with files. Most of the applications with file uploading lets users upload images in order to display on a page, or files which can later be linked for downloading on a page. We believe most of the file uploads are not actually processed by the application, so creating bogus files would only affect the web server or the browser of a visitor. Fuzzing of file uploads might certainly be an area to look into, but we have not done so, as this generally requires writing a separate fuzzer to create the files.

5.3 HTTP Client

A web application takes most user input through a form, and when the user submits the contents of that form, the web browser issues a HTTP GET or POST request with the form contents as parameters. Thus in order to supply fuzz data as input to an application, we need to include a simple HTTP client. This client will be used to send input to the application, and return the resultant response to our fuzz program. Our program can then process the response in order to

find out if it is erroneous.

We have identified the bare minimum that a client must support in order to make fuzzing web applications feasible. A client that implements this functionality is a good start — picking a more full featured client might be better, but make sure the client is easy to program or automate. As an example, we could have based our work on making our fuzzer invoke the `curl` program on the command line, or better: using the `libcurl` library[4]. For convenience, we decided to go with the RFuzz HTTP client, as we are already using this library for other parts of the fuzzer.

The functionality we need from an HTTP client is the following:

1. Easy interface for creating GET and POST requests (without this, it could hardly be called an HTTP client).
2. Possibility to read headers in the response (the easier to access, the better).
3. Possibility to add or modify headers in the request. This is useful to set headers that might be needed to get the request processed correctly. If the client is missing cookie handling (see below) this could be implemented by reading response headers and modifying subsequent request headers. The easier it is to access to the headers, the better for the programmer implementing the fuzzer.
4. Handling of cookies. This isn't strictly necessary, as it might be implemented on through access to headers, but if the client provides it the programmer saves some tedious work. Cookie support through some means is absolutely necessary in order to fuzz pages "hidden" behind a login page.

5.4 The Fuzzer — Tying it all Together

In our application, we have a class called Fuzzer, which is responsible for tying the components together in order to mount the attack. When initialized with a Target (which is created from the attack script), it creates the directory containing the log files, and instantiates a logger for the current session. The logger is used for logging events to a log file as we go, and could be configured to provide logging to the console if necessary. Before starting, we walk through the list L found in the Target, where each element in L consists of a list with method, path and query. Going through L, we evaluate the fuzz tokens found in path and query. This way we get less overhead (even though it is probably small enough) when the actual fuzzing is happening. Recall from Section 5.2 that in order to evaluate the token, all we have to do is call the to string method, as this will in turn call the fuzz method which yields the result. Thus the mapping $L \mapsto L'$ can be achieved simply by calling to_s on the path and the query, like shown in Listing 3.

After evaluating the tokens, the fuzzer starts firing requests based on the information in L'. First it logs the request which it is about to make through the logging facility, then it checks which HTTP method to use. If the method is POST, it takes care to add a header specifying a content type of `application/x-www-form-urlencoded` and puts an urlencoded version of the query in the request body as per Section 17.13.4.1 of the HTML 4.01 specification [33]. If the method is GET, the query is passed as a part of the URL. Note that the RFuzz library urlencodes the query,

[4] Stenberg, D.: cURL and libcurl, http://curl.haxx.se/

```
def evaluate_fuzztokens_in_list(list)
  list.map do |method, path, query|
    # call to_s on path
    path = path.to_s
    query = query.inject({}) do |h,(k,v)|
      val = v.kind_of?(Array) ? v.rand : v
      # call to_s on key and value in query
      h.merge({k.to_s => val.to_s})
    end
    [method, path, query]
  end
end
```

Listing 3: Evaluating FuzzTokens in a list of method; path and query.

but not the request body (as this can be sent with other encodings). For more on urlencoding, please refer to RFC 1738 [41], and the newer RFC 3986 [42].

Having prepared the request, it uses the HTTP client to send it to the host specified in the Target (through the attack script). When the response is received, it logs the status code through the logging facility, records the status code and request timings, and logs a serialized version of the request and response to a YAML[5] file. When all requests have been made, it creates one CSV file containing the recorded number of different status codes, and one CSV file containing statistics on the request timings.

[5]YAML Ain't Markup Language — http://www.yaml.org/

6 Using the Fuzzer

This chapter describes how to use the fuzzer we implemented in the previous chapter. After giving a short intro to how you set up a web application, Section 6.2 will explain how we use the crawler to identify inputs to the application, and how we adjust the outputted scripts to do something more useful. Section 6.3 explains how to invoke the fuzzer with an attack script, and what output to expect. Lastly, Section 6.4 gives examples on how you might analyze the resulting output.

6.1 Set up Target

First, you need to set up the target you will be fuzz testing. This involves installing a web application on a test server, and will in many cases be a trivial job. Most open source applications come with a file called `INSTALL` which gives detailed install instructions. Many PHP applications also feature a web-based installer, which takes notice if it hasn't been set up properly and guides you through the steps to making it work (like e.g. Wordpress). If the application uses a database (as most do these days), a good tip will be to back up that database through `mysqldump` or a similar program. This way you can easily restore the initial state of the application if that is necessary in order to replay the test case.

6.2 Creating the Attack Script

After setting up the application, you need to tell the fuzzer where it can send it's requests, and which parameters it can send. This can be done in many ways, but here we will describe the actions taken in this study. We did this in two steps.

First we used our crawler to crawl the web pages of the target application. Every page it came across was then fed to a callback function which parsed the HTML and ferreted out the forms. Each form was then used to create a possible HTTP request which was written to the screen. The output from the crawler was redirected to a file which we later was to feed to the fuzzer.

Having crawled the site, the attack script had to be manually adjusted. Since the crawler didn't keep a cache of forms it had previously seen, some of the forms had been duplicated (often the case for search fields). The arguments to the request also had to be filled in properly, as the crawler only passed the values which were suggested on the web page. As an example, consider the following: the crawler encounters a web page with the form found in Listing 4.

```
<form action="/search" method="post">
  <input type="text" name="q" value=""/>
  <input type="submit" />
</form>
```

Listing 4: Example HTML form.

The entire document is passed to the form scraping function, but only the form is relevant here. The crawler would then output a section looking like Listing 5.

```
attack("/Welcome_to_Junebug") do
  many :post, "/search", {"q" => ""}
end
```

Listing 5: Example output from crawler after parsing form in Listing 4.

From the output we can see that on the page with URI `/Welcome_to_Junebug`, the crawler found a form that submits to the URI `/search` and which has a single input field with the name of `q` and an empty default value. Going through the output of the crawler later on, we might change it to something looking like Listing 6.

```
attack("Search box") do
  many :post, "/search", { :q => str(100)  }
  many :post, "/search", { :q => byte(100) }
  many :post, "/search", { :q => big       }
end
```

Listing 6: Example of manually tweaked attack script from Listing 5.

When we now choose to run the fuzzer, it will attack the search box in the following way:

1. Send "many" HTTP POST requests to `/search`, with the parameter `q` set to a random string of length 100.
2. Send "many" HTTP POST requests to `/search`, with the parameter `q` set to a random byte sequence of length 100.
3. Send "many" HTTP POST requests to `/search`, with the parameter `q` set to a random big number.

While the manual labour might sound tedious and boring (and it is), we deemed it sufficient for our initial testing. We have proposed ways to improve this part in Chapter 10.

6.3 Running the Fuzzer

Having created and tweaked the attack script, running the fuzzer is as easy as starting the application with the script as the argument: `ruby fuzz.rb targets/my_attack_script.rb`. While the fuzzer runs it will only output some information on the progress to the screen. However if you run `tail -f` on the log file, you can see a more verbose transcript of what's happening. The log file is created in `output/TARGET-NAME/`, where TARGET-NAME is the name specified after `setup` (see Listing 1 for an example). The log file is named with the timestamp of the invocation of the fuzzer.

When the fuzzer is done, the log directory will contain four files:

timestamp-counts.csv A comma separated file containing the counts of various HTTP status codes (and exceptions thrown).

timestamp-runs.csv A comma separated file containing statistics about the timings. Average, max, min times of the requests etc.

timestamp.log The log of events.

timestamp.yaml A serialized version of the requests and responses. The requests are logged as a full string, while the response is logged as a hash of headers along with a string containing the request body (if it exists).

6.4 Aftermath: Analyzing the Results

Analyzing the results is hard to automate, since there are various ways to look at the data to determine what can be considered an erroneous response. However, we recommend starting by looking at the responses where the status code is in the 500 range. Recall from Section 3.4 that a status in the 400 range indicates an error on our part, and a status in the 500 range indicates an error on the server. Knowing that our requests mainly consists of "garbage", the fault should likely be on our part, so in an ideal world, we should thus be certain that if a fuzzed request resulted in a status code in the 500 range, we discovered a flaw in the application or web server (we will later see that this isn't necessarily the case).

We wrote a simple GUI to help filtering responses and show us the headers and response body to make this task even easier. Some of the applications will throw a stack trace at the user (maybe depending on running it in development or production mode) while others will state that an error has occurred and that more is to be found in the server logs. Combining the stack trace with the source code often provides what you need to find the wrong assumptions made by the developers. It might also be worthwhile to check the responses with a status code of 200. As 200 means OK, and you are mostly throwing random "garbage" at the application, some errors might also exist within these responses.

By looking at the `counts.csv` file, you should also be able to see if exceptions are raised. As an example, seeing `ErrnoECONNREFUSED` means that a connection to the web server could not be made. If this occurs after a seemingly OK request, it might mean that one of the previous requests managed to halt the web server. In that case you should check the log to find out which requests have no response. The log contains lines looking like this:

```
I, [2008-02-27 14:28:05#20566]  INFO -- : client => get /rt/Search/Simple.
    ↪html {"q"=>"142370278"}
I, [2008-02-27 14:28:21#20566]  INFO -- : client <= HTTP STATUS: 200
```

First a request is sent from the client, with the value of q being 142370278. The server then sends a response to the client (where only the status code is logged in this file, see the YAML file for more info). If there are two or more subsequent requests (two or more lines matching: "client => "), this means that a response was not received properly.

Using `counts.csv` you can also graph the status codes the requests generated, and using `runs.csv` you can plot the request timings of the runs.

7 Experiment

This chapter explains how we conducted our experiment. Section 7.1 describes the environment in which the project took place, and gives a list of computers and software used. Section 7.2 gives a brief overview of the applications we tested, while more information can be found in Chapter A in the appendix. Finally, Section 7.3 contains information about the results we got during the experiment.

7.1 Environment

The tests have been conducted on two machines, one web server and one attack machine (see Table 3). The web server is borrowed from the IT department at Gjøvik University College, while the attack machine is the author's laptop. The following software has been used on the server (the version numbers match the ones in Debian 4.0 at the time of writing):

- Apache 2.2.3
- PHP 5.2.0-8+etch10
- MySQL 5.0.32
- Ruby 1.8.5
- Perl 5.8.8

On the attack machine the following software has been used:

- Ruby 1.8.6 — The standard version in Mac OS X 10.5
- RFuzz 0.9
- Hawler 0.1
- Hpricot 0.6

	Web server	Attack Machine
Brand	Cinet Smartstation 200	Apple iBook G4
CPU	Pentium III @ 870 MHz	PowerPC G4 @ 1.33 GHz
RAM	377 MB	1.5 GB
Operating System	Debian GNU/Linux 4.0	Mac OS X 10.5

Table 3: The computers we used in the experiment

While testing, the machines were connected through a network cable, using an ad-hoc network with only the attacker and the server present. This way we remove the possibility of other computers interfering with our test environment, without having to set up a dedicated test lab. The server had a monitor and keyboard connected, so by running the `tail` command on the log files, we could inspect what was going on on the server while running the attack script on the attack machine.

7.2 Applications Tested

The following is a list of the applications we have tested during the writing of this thesis. Descriptions are taken from the project pages of the respective applications.

Chyrp "a blogging engine designed to be very lightweight while retaining functionality. It is driven by PHP and MySQL (with some AJAX thrown in), and has a pimpin' theme and module engine, so you can personalize it however you want." We are using version 1.0.3.

eZ Publish "an Enterprise Content Management platform with an easy to use out of the box Web Content Management System. It is available as a free Open Source distribution and serves as the foundation for the rest of the eZ Publish Product Family." We are using version 4.0.0-gpl.

Junebug "a minimalist wiki, running on Camping." We are using version 0.0.37 with Camping 1.5.

Mephisto "a kick ass web publishing system. It's a blog engine with some simple CMS-ish concepts (sections, pages), a very flexible templating system, and an aggressive caching scheme that takes advantage of your web server's best traits." We are using version 0.7.3 with Rails 1.2.3.

ozimodo "a Ruby on Rails powered tumblelog. It's like a blog, but different." We are using version 1.2.1 with Rails 1.1.4.

Request Tracker "an enterprise-grade ticketing system which enables a group of people to intelligently and efficiently manage tasks, issues, and requests submitted by a community of users." We are using version 3.6 (the one included in Debian 4.0).

Sciret "an advanced knowledge based system. In the further development, Sciret will be extended to a full helpdesk system which will also include a trouble ticket system, document management, bookmark management and more." We are using version 1.2.0-SVN-Release-554.

Wordpress "a state-of-the-art semantic personal publishing platform with a focus on aesthetics, web standards, and usability." We are using version 2.3.2.

Amongst these applications we find two applications in use at Gjøvik University College, namely eZ Publish and Request Tracker. eZ Publish is the CMS system used to manage the school's main web site, and Request Tracker is the system used by the IT department to track support requests from students and employees.

Chyrp	eZ	Junebug	Mephisto	Ozimodo	RT	Sciret	Wordpress
$\surd$	$\surd$	$\star$	$\otimes$	$\star$	$\otimes$	$\surd$	$\otimes$

Table 4: Results from applying our fuzzer.
Legend: $\surd$: OK. $\otimes$: Found bug. $\star$: 500 errors.

7.3 Outcome

This section outlines the results we found after having applied the fuzzer as per instructions in Chapter 6. A brief overview is given in Table 4. We will not go into details about Ozimodo and Junebug. In stead, we give a short explanation here. The errors reported on Ozimodo is caused by a failure to log in by the fuzzer, and hence many of the requests resulted in a status code of 302 (for redirection to the login page), 400 (for a user error), but some resulted in 500 (server error), because of a mistake in the redirection layer. We did not investigate this further, as we rather spent time fixing the cookie handling in our fuzzer. The errors reported for Junebug were interesting, but few and hard to track down.

7.3.1 No Server Side Validation of Input

When receiving input from the user, it is important to check this input before letting it propagate through the code. In a web context these checks can be done both on the client side, and on the server side.

Here is a (admittedly) bad example from Mephisto. When creating a new article, the input form contains several dropdown boxes allowing you to pick a date for the article. These dropdown boxes are named `article[published_at(di)]` where $d \in \{1, 2, \ldots, 5\}$. When the user submits the form, the Rails application receives a hash (a keyed collection) with the user input, available through `params` in the code. By naming the dropdown boxes with brackets, the data becomes available in a hash. `params[:article]` will yield a new hash:

```
{:published_at(1i) => "2008", :published_at(2i) => "03", ...}
```

Mephisto contains the following code to convert the information submitted into a date object (Listing 7):

```
def convert_times_to_utc
  with_site_timezone do
    date = Time.parse_from_attributes(params[:article],
                                      :published_at, :local)
    next unless date
    params[:article].delete_if { |k, v|
        k.to_s =~ /^#{:published_at}/
    }
    params[:article][:published_at] = local_to_utc(date)
  end
end
```

Listing 7: Mephisto's method for converting user input to a date.

This code forwards all the input parameters with a name like `article[name]` (in the HTML file), along with two symbols (`:published_at` and `:local`) to `Time.parse_from_attributes`, which is defined as follows (Listing 8):

```
class Time
  class << self
    def parse_from_attributes(attrs, field, method=:gm)
      attrs = attrs.keys.sort.grep(/^#{field.to_s}\(.+\)$/).
        map { |k| attrs[k] }
      attrs.any? ? Time.send(method, *attrs) : nil
    end
end
```

Listing 8: Creating a date through a hash of integers.

This method will first sort the keys in the `attrs` hash, and select only those matching the argument given in the field parameter. Then it will use those keys to look up the values found at those keys in the hash. Finally, it will use the method given as a parameter, and invoke it on the Time class, using the values filtered out as arguments. It might not be apparent immediately, but there are several problems here:

1. The code in `Time.send(method, *attrs)` might raise an exception. This is the case we discovered through our fuzz testing, and the exception is not caught anywhere in the application, as it assumes no user would bother altering the input values in the dropdown boxes. By passing a value of zero for all the `published_at fields`, the code will try to create a new date object as follows:
   `Time.send(:local, *[0, 0, 0, 0, 0])`
   `Time.local` expects it's first argument list to contain (in the following order) year, month, day, hour, minutes, seconds and microseconds (note that everything but the year is optional). This will raise an `ArgumentError` exception, as the second and third fields does not contain valid values for a month and day.
2. An attacker could also send more arguments than intended through the code. Since the code only filters out the input values matching the regular expression /^#{field.to_s}.+$/, we could easily submit fields named "published_atSOMETHING", as this still would match the expression. In this example, we don't gain anything by sending other fields, as they will be passed to `Time.local`, and the greatest harm we can do is to create an erroneous time for the article, or raise an exception. However, if this coding practice is used without care other places, the consequences could be greater.

7.3.2 Incorrect Use of HTTP Status Codes

Our testing showed that some applications used the HTTP status codes in an incorrect way. While testing the comment forms in Wordpress, we noticed that several of our requests resulted in a status code in the 500 range, namely "500 Internal Server Error" (see Figure 7). Inspecting the message body of the response, we found that Wordpress was complaining about the email field not containing a valid email address. In other words, the application had correctly found an error

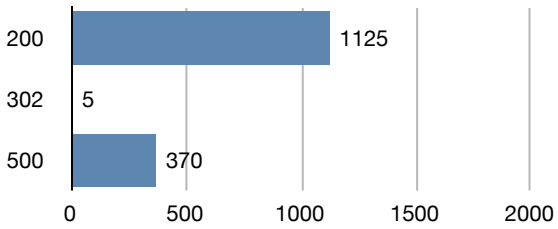

Figure 7: Number of HTTP status codes returned while fuzzing Wordpress.

in the user supplied input, but used a status code indicating that the error was on the server's side. A similar situation appeared if we omitted the email or author field for the comment form.

Also, while fuzzing the login page, all requests were greeted by a response with an 500 status code. However this still didn't mean that the application failed in any other way than using the status code in a semantically correct way. The message body stated that a wrong username or password was used. This means that the application did a successful lookup in the database, but no rows were returned. If the authors of the application had thought this situation through, they should probably have issued the same message body, but with a 401 or 403 status code. Which code should be used is however not entirely clear. The 401 status code can only be given with a `WWW-Authenticate` header, which indicates that HTTP authentication should be used. Most web applications use their own authentication mechanism, and not the one provided by the HTTP protocol, so the 401 code is probably not the right code to use. The 403 status code indicates that "authorization will not help and the request SHOULD NOT be repeated", but if we interpret this to mean HTTP authentication, and not the authentication mechanism used by the web application, it might be the right response to use.

We filed a bug with the Wordpress developers about this issue which can be tracked at `http://trac.wordpress.org/ticket/6076`. See also Section B.1 in the appendix.

7.3.3 Failure to Handle Exceptions

A concern we also noted in Section 7.3.1, is the failure to handle raised exceptions. In that section, we inspected the code handling the creation date of new articles and found that the `Time` class might throw an exception. That exception could easily be avoided by performing simple checks on the user input. If such an error occurs through the use of a form where the user have other fields where longer text can be entered — like a `textentry` or an `input` field — the consequence could be that the user loses his work. Consider a user writing a long article in the web interface, only to find out that after submitting it he gets an exception thrown in his face. The article is not saved on the server because an error occurred, and if the user is using a browser that does not store the input entered, navigating back to the previous page will not work. The author of this thesis has lost an article or two this way himself, and knows the pain associated with having to write it one more time. While the error in Section 7.3.1 is somewhat constructed, we found a similar result in a textarea on the same page.

Many CMS systems uses a "WYSIWYG"[1] editor implemented in JavaScript, but while this lets

[1] What You See Is What You Get

users write formatted articles without knowing HTML, the resulting HTML is quite often more complex than necessary not to mention that it might not work properly in all web browsers. For writing articles, Mephisto tries to make formatting easier by letting the user choose an input filter in stead of a WYSIWYG editor. The input to the textarea where we discovered the bug ended up being sent to such a filter. These filters parse simpler markup to HTML and only requires the user to know a small set of formatting rules. This also gives the document much less "noise" than plain HTML, and since the markup is minimal, it is easy to generate semantically correct HTML. Examples of this are Textile[2] and Markdown[3]. The Ruby library used for handling Markdown formatted text — BlueCloth — is explained further in a book by Berube [43].

As with the date input in the previous section, the text for the article is sent directly to the filter the user has chosen, and while this filter can raise an exception, Mephisto does nothing to handle this case. The text is passed straight to `MarkdownFilter.filter`, which looks something like this (Listing 9):

```
class MarkdownFilter < Base
  def self.filter(text)
    if Object.const_defined?("BlueCloth")
     BlueCloth.new(text.gsub(%r{</?notextile>}, '')).to_html
    else
     text
    end
  end
end
```

Listing 9: Passing input to the Markdown filter.

A problem with passing the text more or less unchecked to a filter is that an attacker can target a vulnerability in the filter in stead of the webapp itself. If there was an exploitable vulnerability in the BlueCloth library, or in one of the other filters, the attack surface is increased. This of course, is a problem that is hard to manage (and a particular issue mentioned on BlueCloth's bugtracker[4] seems very hard to avoid). As we can see in Listing 9, Mephisto does some checking before passing the input along, and as long as the filter does not allow code execution in any form, that problem is avoided.

We see that the code passing input to the filter removes the `<notextile>` tag, along with it's closing tag. This ensures that a user cannot enter plain HTML, as all brackets are automatically converted to their respectable HTML entities. This combats insertion of annoying HTML, as well as potentially malicious JavaScript code. The problem is that the calling code does not catch any exceptions which might be raised, like the `BlueCloth::FormatError` which is raised when the source text contains invalid markup, like unmatched quotes. The possible outcome of this — losing entered text — is explained above. This issue is mostly an annoyance, but we believe this should be fixed, so we informed the developers: http://groups.google.com/group/MephistoBlog/t/d1204a0ad9dd36eb. See also Section B.3 in the appendix.

[2] http://textism.com/tools/textile/
[3] http://daringfireball.net/projects/markdown/
[4] http://www.deveiate.org/projects/BlueCloth/ticket/15

```
Stack:
  [/usr/share/request-tracker3.6/lib/RT/Tickets_Overlay_SQL.pm:240]
  [/usr/share/request-tracker3.6/lib/RT/Tickets_Overlay_SQL.pm:485]
  [/usr/share/request-tracker3.6/html/Elements/TicketList:126]
  [/usr/share/request-tracker3.6/html/Search/Results.html:56]
  [/usr/share/request-tracker3.6/html/Search/Build.html:797]
  [/usr/share/request-tracker3.6/html/autohandler:279] (/usr/share/request-
      ↪tracker3.6/lib/RT/Tickets_Overlay_SQL.pm:487)
```

Listing 10: Apache's error.log

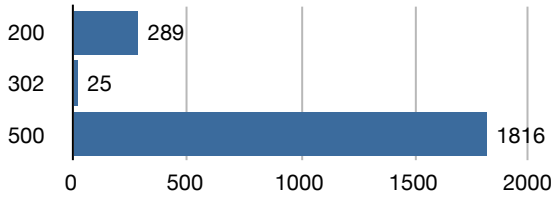

Figure 8: Number of HTTP status codes returned while fuzzing RT.

7.3.4 Resource Exhaustion

One particularly bad bug which we discovered during our fuzz testing was a bug in RT where the application was trying to validate the input. The input was intended to be used for making a query into the ticketing system in order to find tickets matching a search term. To combat attempts at SQL injection or remote code execution, the RT developers have created a parser to ensure the validity of the input before actually using it in the search query. The parser consists of some regular expressions to match the input against known keywords, and a simple state machine specifying which transitions are allowed (see the _parser subroutine in `Tickets_Overlay_SQL.pm`[5]). Somewhere in the parser, we triggered what seems like an infinite recursion.

The bug was triggered early, and we noticed the fuzzer was waiting a long time for a response that never seemed to come. Checking the terminal on the web server, we discovered that the system was running under high load, and performing simple tasks like logging in on a second console took several minutes. Using the `top` program, we could see that Apache was using most of the CPU time, and memory consumption was steadily increasing. Checking the Apache error log, we found a stack trace from RT (Listing 10), which indicated that something indeed had gone wrong. We decided to let the process run in order to see what happened.

After running for about an hour, the operating system kernel stepped in and killed the MySQL process, as all the memory was consumed. Shortly after, a process called `mysqld_safe` restarted MySQL. This two step routine continued several times, until the `mysqld_safe` process failed to start a new MySQL process due to a failure in the pthread library (no new thread could be created, this about three hours after the first stack trace). At last the kernel killed one of the Apache processes, but as Apache is running five child processes to dispatch incoming connections

[5] svn://svn.bestpractical.com/rt/branches/3.6-RELEASE/lib/RT/Tickets_Overlay_SQL.pm

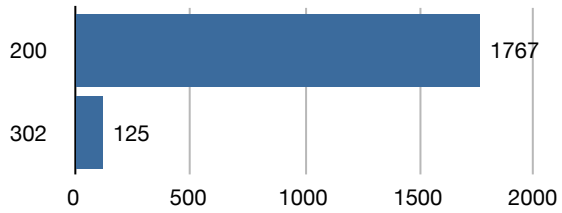

Figure 9: Number of HTTP status codes returned while fuzzing RT, using different seed.

to, and the kernel picked the wrong process, the connection with the fuzzer was still open, so we decided to restart Apache. At this point we were not aware that the MySQL process wasn't running, so the rest of the requests from the fuzzer resulted in a long range of HTTP 500 status codes, as RT could not connect to the database (see Figure 8).

Later we ran the test again, using a different seed (in order to obtain different values for the fuzz tokens), and the resulting status codes were changed dramatically, as we can see from Figure 9.

Since we are not too familiar with Perl, and the bug seemed to be hard to pinpoint, we didn't look further into it. Instead we submitted a bug report to the RT bug tracker. The e-mail correspondence can be seen in Section B.2 in the appendix.

8 Contributions

In this chapter we will present the contributions from this master thesis. We have identified different kinds of contributions: a method for fuzzing web applications, a toolchain implementing this method and findings from the fuzz testing. We also present an overview of how well our method seems to work. Each kind of contribution is presented in it's own section below.

8.1 Method for Fuzzing

The method for fuzzing applications has already been established by Miller et al. This method of testing assumes that programmers makes wrong assumptions about user input, and application of fuzzing against command line and GUI applications has proven to yield results. In the command line example they were able to discover several possible buffer overflow vulnerabilities, and since they had the source code at their disposal they were able to determine the reasons for this.

This project contributes little to the main idea about how to conduct these tests — it is more an evaluation of how well it applies to testing of web applications. Creating a fuzzer has given us a clearer picture about what features it needs to have in order to be efficient to use. For instance, being able to handle cookies is a must in order to fuzz stateful web applications, and providing a means for the fuzzer to "log in" to a website is the key to fuzz the administrative part of the application (or the part of the application available to a privileged user) as well.

8.2 Toolchain for Fuzzing Web Applications

Through this project we have built a toolchain for fuzzing web applications. We have had some building blocks to work with, namely the RFuzz library, the Hawler library and the Ruby programming language. RFuzz was initially created for the task of fuzz testing the Mongrel web server, commonly used by Ruby on Rails applications. The details about this toolchain is described in Chapter 5.

The toolchain we have implemented has some minor similarities to unit testing. You can specify some global state, like hostname and port to use, and specify a "fixture setup" to be executed before each test case — requests from the before block being added before requests from the attack blocks in our case. As with unit tests, the scripts can be written manually or generated in a (semi) automatic way. Unlike unit testing, we have not added a way of asserting results from the requests, so analyzing results will have to be done manually.

8.3 Types of Bugs Found

This section contains an overview of the discoveries we made during the fuzz testing. Details are in Section 7.3.

No server side validation of input (Section 7.3.1) It is our belief that user data should be sanitized before being allowed to propagate through the code. You can never trust a user to enter legitimate values, even if the possible values are "limited" by a dropdown box. As we have seen it is easy to bypass these restrictions. Similarly, using JavaScript to validate user input should only be considered a convenience for the user — not a security measure. Knowing how easy it is to disable JavaScript support in a web browser, we should always enforce the same checks server side as we hope to achieve at the client side. The example provided in Section 7.3.1 might not be a good one, but it still shows that the developers assumptions not always are correct.

Incorrect use of HTTP status codes (Section 7.3.2) While this is not really a security related bug, it is a violation of the semantics described in the HTTP protocol (RFC 2616 [32]). The biggest problem for us is that it makes automating the analysis harder, as we cannot rely on HTTP status codes to tell us how the web server and/or application perceives the error. As we stated in Section 6.4, we should, by the semantics of HTTP 1.1, be able to assert that a status code in the 500 range indicates problems on the server. Not, as was the case with Wordpress, that the application has correctly identified that the problem originates from the user.

Failure to handle exceptions (Section 7.3.3) We saw that Mephisto failed to handle an exception that was raised in a third part library, which in the earlier days of web browsers could mean that all text the user typed in was lost. Most browsers today save the contents of forms in the history (at least for the current session), but a simple formatting error should be caught by the application and not result in showing the user a stack trace they usually don't understand. The programming language used, Ruby, is a dynamic language, and doesn't enforce the programmer to catch an exception or explicitly state that the exception could be thrown as in, say, Java. This might be the reason why these mistakes are easier to make in dynamic languages that enables rapid prototyping.

Resource exhaustion (Section 7.3.4) This type of bug usually manifests itself by causing increased response times and possibly no response at all. This can be caused e.g. by non-terminating recursion and infinite loops. In RT, we discovered what seems to be a non-terminating recursion, resulting in high cpu consumption and a memory leak, followed by a forced process termination. Improper use of recursion can easily lead to this condition, as each time the function is called, return address and new local variables are put on the stack. Failing to terminate the recursion will thus lead to exhaustion of memory.

9 Discussion

This chapter provides a discussion of the discoveries we made during the fuzz testing. We start off by discussing the completeness of our method, and how this could be improved. Section 9.2 discusses how the results from the different applications compare to each other and Section 9.3 tries to look at how programming practices — or the lack of them — affects the applications we have tested. At last, Section 9.4 tries to compare our approach to the tools mentioned in Section 2.2.6.

9.1 Completeness of our Method

While this method has potential for use in discovering vulnerabilities in web applications, there are some problems with regards to the completeness of the method. As noted in Section 5.1, our crawler did not contain logic to prevent it from logging out from the administrator interfaces of the web applications it was crawling. This resulted in a varying degree of completeness with regards to testing of the applications. Since the crawler is using a breadth-first approach, and most admin interfaces have a logout link on every page, the degree of completeness depends on how many links there are on the initial page behind the login page, and where in the document the logout link is placed. If there are few links on the page, or if the logout link comes early, the coverage of the application will be bad. We propose a solution to this problem in Chapter 10.

Since this thesis mainly has been an evaluation of fuzzing for web applications, we have not tested the applications as thoroughly as one should do when testing applications before putting them in production use. We are mainly concerned with finding out if the method can be applied to web applications, and finding some errors are sufficient for us. For a real evaluation of a product, more comprehensive tests should be done. This can be achieved by writing more complete attack scripts, improving the crawler, trying the same attack script with different seeds, and of course increasing the amount of repetitions done. Section 7.3.4 is evidence that short tests not necessarily yields results. The first run we did was using the word “CHANGEME” as seed and revealed a severe bug, while running the same script with the intended seed: “RT” yields no results.

9.2 Comparability of Results

The most serious bugs were found among the applications with the highest lines of code count (see Appendix A for details). Mephisto and RT are placed at second and fifth accordingly. Wordpress, in which we found a semantical flaw, but not a bug per se, is placed at fourth. This could indicate that bigger applications are more likely to contain bugs which might be found using fuzz testing. However, for the reason mentioned in Section 9.1 this is hard to prove. According to that theory, we should for instance have found bugs in the other big applications. eZPublish has a code base of approximately three times the size of Mephisto (again, see Appendix A), and should — in theory — contain bugs as well, but no bugs were found. One of the reasons for this

could be the lack of completeness in our method (Section 9.1). We had several tests directed at the user interface in eZ, but none at the administrative interface. The same holds for Sciret (which is placed third in application size). In order to draw better conclusions about application size and probability of finding bugs, the method needs to be more complete.

As the bugs we found in Mephisto were discovered in the administrative interface, one might argue that they are less significant than the ones found in the user interface. This is of course true to a certain extent. As Mephisto is a CMS system, chances are that some sites deploy this system with a large amount of users. Only one account needs to be compromised to get access to the administrative interface, and while some CMS systems enforce user levels, the attack surface will still be increased by compromising an account.

9.3 Programming Practices

Many people claim that good programming practices and "safe coding guidelines" should be used in order to make programs more secure. While this might work, it is human to err and all too easy to make mistakes even though you in your heart know how it should be done. A good example of this is when Bowers et al. [17] discovered a bug in the original fuzzer from Miller et al. Buffer overflows are still a big problems with C programs, even though the problem has been known for a long time, and several guidelines on how to avoid these problems exists. The best way to avoid buffer overflows is to use an environment where they do not exist, as mentioned in [44] (see e.g. it's Section 3.3.4). This article also mentions other ways to defend against buffer overflows.

Fuzz testing will give an indication of how well the developers validate the user supplied data. While some programming languages can provide "tainting" of input, and requires that this input must be validated in some way before being allowed to propagate through the code (Perl and Ruby has this functionality), it doesn't seem to be used in the web frameworks. We have previously stated that every application that returns a HTTP status code 500, could be considered as an application that doesn't validate or handle user supplied data in a correct way, as all fuzz data we deliver is user supplied data, and that the error clearly is on the user side.

There is a fifty-fifty split between applications who relies on a web framework, and applications that "rolls their own." Mephisto and Ozimodo is using Rails[1], Junebug is using Camping[2] and Request Tracker uses Mason[3] for HTML layout. The others (Chyrp, eZ Publish, Sciret and Wordpress) are written in plain PHP, and while some of them uses a Model-View-Controller pattern (first introduced by Reenskaug [45, 46]), it is implemented by the developers of the application, and not provided by a generic framework.

Lately, some emphasis has been put on testing software using unit tests, but even though the Rails applications have a large amount of unit tests (as did the Camping application), these are not sufficient for avoiding bugs. We discovered bugs in Mephisto, and some minor bugs in Junebug which we have not discussed, as we are not entirely sure what went wrong. The Ruby applications seems to be the only applications among the tested which makes use of unit tests.

[1] http://rubyonrails.org
[2] http://code.whytheluckystiff.net/camping
[3] http://www.masonhq.com/

Program	Automatability	Documentation	Ease of use	Feasibility for web apps
Burp Suite	Poor	Good	OK	Good
fuzz.rb	Good	N/A	Good	Good
Peach	Good	Poor	Poor	OK
Sulley	Good	Good	Good	OK

Table 5: Comparison of fuzzing tools

9.4 Comparison

Here we will present a short description on how our fuzzer and the alternatives presented in Section 2.2.6 compares to each other, and how they differ in use. A summary is shown in Table 5. Please note that this section is based on our own judgement.

We believe the Burp Suite is harder to automate than the alternatives, as it requires manual intervention at many phases. The analyzer provided is OK though, and at this stage in fuzz testing manual intervention is usually required anyway. The Burp Suite might be a good alternative if you're looking to do more than just fuzzing (like e.g. enumeration attacks) from the same tool.

Looking at the Peach Fuzzing Platform, we found that the provided tools are "not quite there yet", and the documentation was also lacking a bit. The realistic way to create a fuzzer for web applications, would be to first implement components to ease the task of creating these fuzzers — not unlike what we have done with RFuzz. We could probably have based our work on the Python API, but having more experience with Ruby, RFuzz was a better choice for getting a prototype going, even though Peach has more features.

Sulley seems very promising, but for fuzzing web applications, we believe our own approach is better. As with Peach, we could have used Sulley as a framework for building our fuzzer. Writing attack scripts for Sulley seemed straightforward enough, but we believe this framework is better suited for fuzzing applications which provide some sort of communication over a network protocol, rather than web applications.

We wrote a simple script to fuzz a web page in order to compare the results with our own fuzzer. Sulley performs many repetitions, so fuzzing a single entry point took quite some time. This is of course a good thing when you are using the fuzzer for evaluating your software, as you want a good coverage of test cases. However, after having run the fuzzer, we found it hard to analyze the results, as no debugger or process monitor had been attached. The only thing logged was a malformed PCAP file[4], so we got no indication to what had happened during the session. Logging the package stream is OK when you are fuzzing protocols, but might be a bit crude when you are fuzzing a web application.

What makes our approach different from the other tools mentioned, is that Peach and Sulley mainly are frameworks for building different kinds of fuzzers (even though they focus on network based fuzzers), while ours is a fuzzer meant specifically for web applications. While the Burp Suite also contains a fuzzer and is aimed at web applications, this is a more general purpose vulnerability discovery tool, while our tool only does one job. We believe that our fuzzer is easier to use for ad-hoc projects and for getting started quick. The biggest problems with our fuzzer is

[4] a file containing a packet dump of the network traffic generated and received by Sulley. Produced by libpcap: http://www.tcpdump.org/

the lack of documentation and the level of completeness (see e.g. Chapter 10).

10 Future Work

This chapter contains proposals for future work — both for improvements of this fuzzer, and for other possible uses for this approach to web application fuzzing.

- We can imagine a similar approach to be effective in fuzzing web services. By parsing a WSDL (Web Service Description Language) file, one might be able to automate attack script creation and / or direct fuzzing of the provided WSDL ports. Along with the new focus on providing web services for integration between web based systems, a focus on testing these services should follow. For RESTful web services, an attack script might have to be written by hand, as these services seldom are described by a unified description language.

- Crawler: Possibility to "blacklist" a page to avoid logging out from administrative pages.

- Crawler: Store method, path and query options for forms in order to avoid the same form appearing several times. For instance: a form used to search for articles in a CMS system will typically appear on multiple pages. We should be able to identify that we have encountered the same form before as the method (GET or POST), path and query options used will be the same. We only need to fuzz this entry point one time.

- To simplify writing attack scripts we could make the fuzzer pick a random fuzz token for all fields with a value of "nil", and let this be the standard value generated by the crawler. This way we only need to fill in the fields where we know we want a random integer or a random string, and let the fuzzer pick something even more random for the other cases. This is a similar approach to the one taken in [29].

- A slightly different approach to entry point discovery and fuzzing could be taken: we could combine the crawler and fuzzer in order to "fuzz as we go". We could develop this as a one-pass or two-pass system. The one-pass system would crawl a page and store links to other pages, and before traversing them, fuzz all entry points on the current page. The two-pass system would crawl the entire application looking for entry points, and yield control to the fuzzer after visiting all pages. In both cases we need to set the seed used for the random number generator. This could be done either through an option to the program — passed on the command line — or by using a combination of the hostname, initial path or other variables we expect to remain static through the testing period, assuring repeatable results.

- In our system, the attack command takes an optional argument `attack("name")` (e.g. line 1 in Listing 6). This argument is not actually used, but we could perhaps make use of it to generate better logs or something else.

- We are not fuzzing file uploads. This might be an area worth looking into. Some applications lets admin users upload templates for web pages or serialized content to be added to the site. This will probably require writing of a file fuzzer.

11 Conclusions

We will now give an answer to our research questions based on the work that has been presented in the thesis. Regarding how much work it is to implement the fuzzer, our prototype was developed by one person over about two months (not full time). Producing the fuzzer is only needed once, as it is flexible enough to use against several web applications. Writing an attack script for an application can take about a day, but as we noted in e.g. Chapter 10, there is potential for improvements here. When it comes to effectiveness, automation of running the fuzzer is easy once the attack script has been written, but automation of analysis is, as stated in Section 6.4, harder. Bugs we have found has been listed and classified in Section 7.3 and summarized in Section 8.3. The number of applications we found bugs in was briefly illustrated in Table 4 (page 35).

The tests we have been running are not comprehensive enough to give us a basis for making bold statements about the quality of the applications we have tested. However, we believe the results we found is a good indication that fuzz testing indeed can be used in combination with web applications. By running relatively few tests we managed to discover several bugs, and some potential bugs which were not investigated fully (mentioned briefly in Section 9.3). As we noted in Section 9.1, more thorough tests are necessary for testing software to be put in production. Bigger test suites with a greater number of repetitions will likely yield more results. The biggest hurdle is to find a good way of analyzing the results. For our purposes, checking the responses with status 500 was good enough, but for bigger result sets, other techniques might be more appropriate, like checking for certain strings in the response body (as e.g. Stuttard and Pinto does [14]).

Our work shows that some web applications indeed are vulnerable to fuzzing. Not only new and fragile applications, but also “tested and true” applications, as well as applications which has been developed with a focus on unit testing.

Bibliography

[1] Miller, B. P., Fredriksen, L., & So, B. December 1990. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12), 22.

[2] Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarjan, A., & Steidl, J. April 1995. Fuzz revisited: A re-examination of the reliability of unix utilities and services. *Computer Sciences Technical Report*, 1268, 23.

[3] Forrester, J. E. & Miller, B. P. August 2000. An empirical study of the robustness of windows nt applications using random testing. *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4 WSS'00*, 10.

[4] Miller, B. P., Cooksey, G., & Moore, F. July 2006. An empirical study of the robustness of macos applications using random testing. *Proceedings of the 1st international workshop on Random testing RT '06*, 9.

[5] van Sprundel, I. 2005. Fuzzing: Breaking software in an automated fashion. 22nd Chaos Communication Congress (http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf). (Visited May 2008).

[6] Ryber, T. *Essential Software Test Design*, chapter Test Design Techniques: An Overview. Unique Publishing Ltd, 2007.

[7] ISO/IEC 9126-1. 2001. Software engineering – product quality – part 1: Quality model. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749.

[8] Black, R. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*, chapter Reactive Testing, 270–271. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[9] Bach, J. apr 2003. Exploratory testing explained. http://www.satisfice.com/articles/et-article.pdf. (Visited June 2005).

[10] Oehlert, P. 2005. Violating assumptions with fuzzing. *Security & Privacy Magazine, IEEE*, 3(2), 58–62.

[11] Lipner, S. 2004. The trustworthy computing security development lifecycle. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, 2–13, Washington, DC, USA. IEEE Computer Society.

[12] Xiao, S., Deng, L., Li, S., & Wang, X. 2003. Integrated tcp/ip protocol software testing for vulnerability detection. *Computer Networks and Mobile Computing, 2003. ICCNMC 2003. 2003 International Conference on*, 311–319.

[13] Su, Z. & Wassermann, G. 2006. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 372–382, New York, NY, USA. ACM Press.

[14] Stuttard, D. & Pinto, M. 2007. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley.

[15] Miller, C., Honoroff, J., & Mason, J. 2007. Security evaluation of apple's iphone. http://securityevaluators.com/iphone/exploitingiphone.pdf. (Visited May 2008).

[16] Miller, C. & Honoroff, J. jun 2007. Hacking leopard: Tools and techniques for attacking the newest mac os x. https://www.blackhat.com/presentations/bh-usa-07/Miller/Whitepaper/bh-usa-07-miller-WP.pdf. (Visited May 2008).

[17] Bowers, B. L., Lie, K., & Smethells, G. J. An inquiry into the stability and reliability of unix utilities. http://pages.cs.wisc.edu/~blbowers/fuzz-2001.pdf. (Visited May 2008).

[18] Ghosh, A. K., Shah, V., & Schmid, M. 1998. An approach for analyzing the robustness of windows NT software. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, 383–391.

[19] nov 1998. The bulletproof penguin. http://pages.cs.wisc.edu/~blbowers/fuzz-2001.pdf. (Visited May 2008).

[20] Hertzfeld, A. *Revolution in The Valley: The Insanely Great Story of How the Mac Was Made*, 184–186. O'Reilly Media Inc., 1 edition, dec 2004.

[21] Johnson, M. K. 1996. Stop the presses. *Linux Journal*, 1996(27es), 12.

[22] Claessen, K. & Hughes, J. 2000. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 268–279, New York, NY, USA. ACM Press.

[23] Kropp, N. P., Koopman, P. J., & Siewiorek, D. P. 1998. Automated robustness testing of off-the-shelf software components. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, 230, Washington, DC, USA. IEEE Computer Society.

[24] Schmid, M. & Hill, F. 1999. Data generation techniques for automated software robustness testing. *Sixteenth International Conference on Testing Computer Software (ICTCS'99)*.

[25] Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., & Vigna, G. 2006. Snooze: Toward a stateful network protocol fuzzer. *Information Security*, 343–358.

[26] Aitel, D. The advantages of block-based protocol analysis for security testing. Technical report, Immunity Inc., 2003.

[27] Kaksonen, R. 2001. Software security assessment through specification mutations and fault injection. *Communications and Multimedia Security Issues of the New Century*.

[28] Granneman, S. jul 2006. A month of browser bugs. http://www.securityfocus.com/columnists/411. (Visited May 2008).

[29] Butti, L. & Tinnès, J. 2007. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*.

[30] Mendonça, M. & Neves, N. F. 14-16 Nov. 2007. Fuzzing wi-fi drivers to locate security vulnerabilities. *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, 379–380.

[31] Sutton, M., Greene, A., & Amini, P. jul 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 1 edition.

[32] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. jun 1999. Rfc 2616: Hypertext transfer protocol – http/1.1. http://www.ietf.org/rfc/rfc2616.txt. (Visited May 2008).

[33] Raggett, D., Hors, A. L., & Jacobs, I. dec 1999. Html 4.01 specification. http://www.w3.org/TR/REC-html40/. (Visited May 2008).

[34] Fielding, R. T. & Taylor, R. N. may 2002. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2), 115–150.

[35] Goldberg, A. & Robson, D. 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[36] McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4), 184–195.

[37] Huang, Y.-W., Huang, S.-K., Lin, T.-P., & Tsai, C.-H. 2003. Web application security assessment by fault injection and behavior monitoring. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 148–159, New York, NY, USA. ACM.

[38] Kaukonen, K. & Thayer, R. jul 1999. A stream cipher encryption algorithm "arcfour". IETF Draft.

[39] Fluhrer, S. R., Mantin, I., & Shamir, A. 2001. Weaknesses in the key scheduling algorithm of rc4. In *SAC '01: Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, 1–24, London, UK. Springer-Verlag.

[40] Harris, B. jan 2006. Improved arcfour modes for the secure shell (ssh) transport layer protocol. RFC 4345.

[41] Berners-Lee, T., Masinter, L., & McCahill, M. dec 1994. Rfc 1738: Uniform resource locators (url). http://www.ietf.org/rfc/rfc1738.txt. (Visited May 2008).

[42] Berners-Lee, T., Fielding, R., & Masinter, L. feb 2005. Rfc 3986: Uniform resource identifier (uri): Generic syntax. http://www.ietf.org/rfc/rfc3986.txt. (Visited May 2008).

[43] Berube, D. *Practical Ruby Gems*, chapter Easy Text Markup with the BlueCloth Gem, 45–51. Apress, 2007.

[44] Cowan, C., Wagle, P., Pu, C., Beattie, S., & Walpole, J. 2003. Buffer overflows: attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, 227–237.

[45] Reenskaug, T. may 1979. Thing-model-view-editor an example from a planningsystem. http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf. (Visited May 2008).

[46] Reenskaug, T. dec 1979. Models - views - controllers. http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf. (Visited May 2008).

A More Information About the Webapps Tested

An overview of the different properties of the web applications tested is given in Table 6. We also present an explanation of the terms used in the table.

Language The primary programming language used to build the application. As you can see from Section A.1, all applications are written in several languages and – being web applications – most of them use HTML, CSS and JavaScript. Due to the size of the JavaScript libraries included in Sciret, it seems it is primarily written in JavaScript, even though the main programming language used for the application logic is PHP.

KLOC Kilo Lines Of Code. Number of lines in the source code for the application divided by 1000. The KLOC counts for the applications are obtained using Ohcount[1]. See complete reports below.

Weblog A web application that eases the task of keeping an online journal. In a weblog, the entries are usually presented in reverse chronological order, i.e. the newest first.

CMS Content Management System. A web application that aims to help the user manage an entire website – usually without requiring the user to know HTML. Often used to manage sites on a company's intraweb as well as on the internet.

Wiki A lightweight CMS, with a focus on easing the process of creating new pages and linking of pages within the system. The word "wiki" is Hawaiian and means "fast". Wikipedia is the most commonly known wiki.

Ticketing system A system for managing support requests. Requests can be assigned a user, the progress can be tracked, and previous support requests can be referenced. A ticketing system is quite similar to bug tracking systems and issue tracking systems. A clear distinction can be hard to make.

[1] http://labs.ohloh.net/ohcount

Application	Purpose	Language	KLOC	Release date
Chyrp	Weblog	PHP	8.6	2007-12-18
eZ Publish	CMS	PHP	668.9	2000-11
Junebug	Wiki	Ruby	9.8	2006-10-22
Mephisto	CMS	Ruby	218.5	2006-08-10
Ozimodo	Weblog	Ruby	20.8	2005-09-22
Request Tracker	Ticketing system	Perl	66.6	1998-01-31
Sciret	Knowledge base	PHP	159.7	2006-11-05
Wordpress	Weblog	PHP	90.5	2003-06

Table 6: Overview of the applications tested

Knowledge base A system for tracking answers to questions. The goal of a knowledge base system is to collect answers to known problems, and is often used to ease the burden of the customer consultants in a company. Knowledge base systems may be created from scratch, or wiki-like systems could be used.

A.1 Source Lines of Code

Chyrp

Language	Files	Code	Comment	Comment %	Blank	Total
php	77	4723	118	2.4%	431	5272
css	17	1608	130	7.5%	119	1857
html	55	1280	0	0.0%	35	1315
javascript	4	88	59	40.1%	12	159
Total	90	7699	307	3.8%	597	8603

eZ Publish

Language	Files	Code	Comment	Comment %	Blank	Total
php	1378	376261	78149	17.2%	46233	500643
html	1659	63242	379	0.6%	12553	76174
css	97	24912	1215	4.7%	6537	32664
javascript	103	18326	10115	35.6%	4352	32793
xml	37	15287	0	0.0%	0	15287
sql	21	5239	6	0.1%	3568	8813
cncpp	10	1653	380	18.7%	220	2253
perl	4	157	14	8.2%	41	212
shell	4	82	11	11.8%	24	117
Total	3188	505159	90269	15.2%	73528	668956

Junebug

Language	Files	Code	Comment	Comment %	Blank	Total
ruby	23	4673	555	10.6%	918	6146
javascript	1	2909	286	9.0%	168	3363
css	1	237	37	13.5%	55	329
Total	25	7819	878	10.1%	1141	9838

Mephisto

Language	Files	Code	Comment	Comment %	Blank	Total
ruby	1978	148377	21965	12.9%	24274	194616
javascript	25	13517	588	4.2%	2191	16296
sql	41	2700	24	0.9%	358	3082
css	10	2188	164	7.0%	407	2759
html	123	1403	74	5.0%	288	1765
shell	1	5	1	16.7%	2	8
Total	2077	168190	22816	11.9%	27520	218526

Ozimodo

Language	Files	Code	Comment	Comment %	Blank	Total
ruby	107	7777	1167	13.0%	1507	10451

javascript	27	4615	222	4.6%	717	5554
html	56	2939	107	3.5%	792	3838
css	8	869	15	1.7%	151	1035
Total	152	16200	1511	8.5%	3167	20878

Request Tracker

Language	Files	Code	Comment	Comment %	Blank	Total
perl	144	29353	8938	23.3%	13812	52103
html	90	9876	83	0.8%	1157	11116
css	27	2223	19	0.8%	208	2450
javascript	7	862	21	2.4%	68	951
Total	268	42314	9061	17.6%	15245	66620

Sciret

Language	Files	Code	Comment	Comment %	Blank	Total
javascript	291	49142	9318	15.9%	8061	66521
html	138	28903	2032	6.6%	3330	34265
php	186	20875	5260	20.1%	2900	29035
cncpp	21	13915	2014	12.6%	2190	18119
css	45	5407	455	7.8%	1046	6908
perl	13	1984	275	12.2%	286	2545
shell	15	689	425	38.2%	198	1312
xml	5	468	75	13.8%	21	564
sql	4	300	47	13.5%	57	404
bat	1	8	1	11.1%	2	11
Total	635	121691	19902	14.1%	18091	159684

Wordpress

Language	Files	Code	Comment	Comment %	Blank	Total
php	210	39683	4390	10.0%	8346	52419
javascript	91	22671	2010	8.1%	4739	29420
css	38	3821	172	4.3%	788	4781
html	109	2955	51	1.7%	839	3845
xml	1	36	0	0.0%	7	43
Total	313	69166	6623	8.7%	14719	90508

B Bug Reports

B.1 Wordpress

```
Ticket #6076 - Incorrect use of HTTP status code 500

Reported by: runeh
Description:

When trying to log in using a wrong username or password, Wordpress
issues an HTTP 500 status code, and the request body states that wrong
credentials has been supplied. As I am sure you know, 500 means Internal
Server Error. A more suitable status code to use would be 403
(Forbidden).

The 500 status code is also issued when submitting an empty username or
password. I think 400 (Bad Request) would be a better fit. Both of these
errors are made by the user, not the server, so it stands to reason that
the status codes should be in the 400 range.

I also noted that a 500 status code is issued if a user is commenting to
quickly (as an anti-spam measure). I'm not convinced that 500 is the
correct status to use, however I'm unsure about which status fits best.

Fixing these issues will make Wordpress behave more semantically correct
with regards to the HTTP protocol.

----------

Changelog:

03/07/08 22:52:42 changed by westi:
* keywords set to needs-patch.
* owner changed from anonymous to westi.
* status changed from new to assigned.

05/06/08 02:51:12 changed by guillep2k:
* cc set to guillep2k.

  Personally, I don't think any status codes different from 200 would be
  correct, since at HTTP level you're actually delivering the requested
  page (i.e., the results of the user providing login credentials). 4xx
  error codes are (again according to my opinion) for HTTP errors (e.g.,
  400/Bad Request is for a malformed HTTP request, like invalid usage of
  headers) and not for application level errors. Using HTTP (transport)
  error codes to reflect application statuses seems to me like mixing
  design layers. Proxies and user agents act upon these errors and may
  choose not to show the user your provided HTTP response body but
  replace it for a body of their own.
```

```
Nevertheless, 500 is obviously wrong. 403 is meant to reflect the fact
that access is forbidden, despite any credentials you may provide (so,
403 is semantically incorrect). The correct code should be 401, but
that's for the HTTP protocol, not the Wordpress application. You know
what happens if you send back a 401 status: the user agent (browser)
will ask for credentials and attempt basic authentication. This is my
point as why I think we're mixing layers if we pretend to use HTTP
codes different from 200.

In my opinion, Wordpress shouldn't return anything except: 200 (pages
served), 404 (custom page not found page) and 303 (to redirect to a
permalink). See a good article about these codes at Wikipedia
http://en.wikipedia.org/wiki/List_of_HTTP_status_codes.
```

B.2 Request Tracker

The initial bug report to the RT bug tracker:

```
From: Rune Hammersland <rune.hammersland@hig.no>
Subject: Query error causing RT to hang
Date: 1. april 2008 15:41:38 GMT+02:00
To: rt-bugs@bestpractical.com

I was doing some fuzz testing on RT, and managed to generate a request
that unveiled a bug. After sending the request, the load on the server
steadily went up. After running for a while (some hours admittedly), all
the memory on the server was consumed. syslog has several references to
"Deep recursion on subroutine", so the memory leak is not surprising.

After digging around in the RT code base, I figured the error lies in
the query parser which is used to find out if the query is well formed.
Sadly I'm not very good with Perl, so that's how far I got.

The offending request was a POST request to /rt/Search/Build.html, with
the POST body given in the attached postbody.txt (correctly encoded as
application/x-www-form-urlencoded), but given as key-value pairs here).

Attached is also an excerpt from syslog and error.log from Apache.

The version of RT used is 3.6.1-4, from Debian 4.0, served by Apache 2.

Steps to reproduce:

* Log in to RT.
* Send POST request to /rtpath/Search/Build.html with the key/value
 pairs from postbody.txt encoded as x-www-form-urlencoded.

--
Best regards
Rune Hammersland
```

The response from Jesse Vincent, asking for a more convenient way to reproduce the bug:

```
From: Jesse Vincent via RT <rt-bugs@bestpractical.com>
Subject: Re: [fsck.com #9122] Query error causing RT to hang
Date: 9. april 2008 18:43:50 GMT+02:00
To: rune.hammersland@hig.no

Rune, I'm sorry for the delay in my reply. I greatly appreciate the
digging at this. Do you have a convenient wget/curl/whatever
invocation to post this stuff encoded correctly?

Also, is the fuzz tester you use opensource? It sounds useful and I'd
love to add it to my toolkit

Best,
Jesse
```

The reponse to Jesse's mail, containing a shell script with curl commands to reproduce the problem:

```
Subject: Re: [fsck.com #9122] Query error causing RT to hang
From: Rune Hammersland <rune.hammersland@hig.no>
Date: 11. april 2008 14:54:28 GMT+02:00
To: rt-bugs@bestpractical.com

On 9. april. 2008, at 18:43, Jesse Vincent via RT wrote:
> Rune, I'm sorry for the delay in my reply. I greatly appreciate the
> digging at this. Do you have a convenient wget/curl/whatever
> invocation to post this stuff encoded correctly?

See the attached shell script which uses curl. Edit the USER and PASS
variables to enable it to log into your application. You might want to
delete the cookie file after running it (as it hangs on the second curl
command).

> Also, is the fuzz tester you use opensource? It sounds useful and I'd
> love to add it to my toolkit

I'm writing a master thesis about fuzz testing web applications, and
wrote my own fuzzer based on the RFuzz library, in the hopes of making
it easier to apply to webapps than other fuzzers are (as they are often
generalized for protocols or filetypes). My intentions were to release
it when done, but the code is getting pretty awful, and I have some
enhancement ideas which probably should be implemented.

I can send you a copy of the code if you like, but I don't think it's
very useful for others than myself at this point. :P

--
Best regards
Rune Hammersland

<ticket9122.sh>
```

Acknowledgement of the bug:

```
From: Jesse Vincent via RT <rt-bugs@bestpractical.com>
Subject: Re: [fsck.com #9122] Query error causing RT to hang
Date: 11. april 2008 15:13:35 GMT+02:00
To: rune.hammersland@hig.no

On Apr 11, 2008, at 8:54 AM, rune.hammersland@hig.no via RT wrote:
> On 9. april. 2008, at 18:43, Jesse Vincent via RT wrote:
> > Rune, I'm sorry for the delay in my reply. I greatly appreciate the
> > digging at this. Do you have a convenient wget/curl/whatever
> > invocation to post this stuff encoded correctly?
>
> See the attached shell script which uses curl. Edit the USER and PASS
> variables to enable it to log into your application. You might want
> to delete the cookie file after running it (as it hangs on the second
> curl command).

Perfect. Just almost took down my workstation with it ;)

> > Also, is the fuzz tester you use opensource? It sounds useful and
> > I'd love to add it to my toolkit
>
> I can send you a copy of the code if you like, but I don't think it's
> very useful for others than myself at this point. :P

I'm happy to wait, but consider me the first of an army of folks
demanding that you release it at the end, no matter the code quality ;)
```

The bug is not triggered in later versions of RT:

```
From: Ruslan U. Zakirov via RT <rt-bugs@bestpractical.com>
Subject: [fsck.com #9122] Query error causing RT to hang
Date: 10. june 2008 14:43:08 GMT+02:00
To: rune.hammersland@hig.no

On Fri Apr 11 09:13:34 2008, jesse wrote:
> On Apr 11, 2008, at 8:54 AM, rune.hammersland@hig.no via RT wrote:
>>
>> <URL: http://rt3.fsck.com//Ticket/Display.html?id=9122 >
>>
>> On 9. april. 2008, at 18:43, Jesse Vincent via RT wrote:
>>> Rune, I'm sorry for the delay in my reply. I greatly appreciate the
>>> digging at this. Do you have a convenient wget/curl/whatever
>>> invocation to post this stuff encoded correctly?
>>
>> See the attached shell script which uses curl. Edit the USER and PASS
>> variables to enable it to log into your application. You might want
>> to delete the cookie file after running it (as it hangs on the second
>> curl command).
>
> Perfect. Just almost took down my workstation with it ;)

On the latest 3.8 worked for me without big recursions, just showed an
error. Can you, guys, try to reproduce?
```

```
--
Regards, Ruslan.
```

But that doesn't necessarily mean it's fixed:

```
From: Alex Vandiver via RT <rt-bugs@bestpractical.com>
Subject: [fsck.com #9122] Query error causing RT to hang
Date: 12. juni 2008 21:59:16 GMT+02:00
To: rune.hammersland@hig.no

On Tue Jun 10 08:43:03 2008, ruz wrote:
> On the latest 3.8 worked for me without big recursions, just showed an
> error. Can you, guys, try to reproduce?

It's still reproducable under 3.6 HEAD, but the same script doesn't
cause 3.8 to explode.  I suspect this is because some parameters got
renamed, and not because the underlying problem has been fixed.
- Alex
```

Finally, the bug is tracked down:

```
From: Alex Vandiver via RT <rt-bugs@bestpractical.com>
Subject: [fsck.com #9122] Query error causing RT to hang
Date: 12. juni 2008 23:34:58 GMT+02:00
To: rune.hammersland@hig.no

This bug is triggered by any code path that causes perl to die() when
the query parameters contain invalid UTF-8 bytes sequences.  When the
die() happens, RT attempts to build a stack track using
Devel::StackTrace.  Devel::StackTrace tries to inspect the arguments to
every subroutine call, which triggers rt.perl.org #41530.  This, in
turn, throws an exception, which generates a stack trace, and so on.

One solution is for Mason to disable its error handler when it is
running, which will at least prevent reentrant loops like this one from
happening.  I am unclear on the correct solution to the underlying
encoding bug.
- Alex
```

B.3 Mephisto

```
From: Rune Hammersland <rune.hammersland@gmail.com>
Date: Tue, 15 Apr 2008 06:08:59 -0700 (PDT)
Subject: FormatError in BlueCloth library

I don't know if this is of any concern, but here goes:

Creating a badly formatted post with the markdown filter obviously
raises an exception. The problem is that it isn't caught, and results
in an ugly stack trace the user shouldn't need to see. Would it be
possible to catch this exception at some point and handle it
gracefully? Maybe returning to the edit page with a flash message
containing the message in the exception?
```

```
Most web browsers stores the content you typed into a form, so for
many users hitting the back button would suffice to get their text
back (or they could indeed get it from the parameters hash at the
bottom of the trace), but I remember the time when that was not the
case.

A simple example of input that will result in a BlueCloth::FormatError
is the following:

This is some ‘code

The exception message in this case would be:

Bad markdown format near "code": No "‘" found before end

PS: This was under Mephisto 0.7.3, but it seems the filters are
handled the same way in trunk (without having confirmed it).

--
Regards
Rune Hammersland
```

Fuzz testing of web applications

Rune Hammersland and Einar Snekkenes
Faculty of Computer Science and Media Technology
Gjøvik University College, Norway
email: firstname.lastnamehig.no

***Abstract*—The handling of input in web applications has many times proven to be a hard task, and have time and time again lead to weaknesses in the applications. In particular, due to the dynamics of a web application, the generation of test data for each new version of the application must be cheap and simple. Furthermore, it is infeasible to carry out an exhaustive test of possible inputs to the application. Thus, a certain subspace of all possible tests must be selected. Leaving test data selection to the programmers may be unwise, as programmers may only test the input they know they can expect. In this paper, we describe a method and tool for (semi) automatic generation of pseudo random test data (fuzzing). Our test method and toolkit have been applied to several popular open source products, and our study shows that from the perspective of the human tester, our approach to testing is quick, easy and effective. Using our method and tool we have discovered problems and bugs with several of the applications tested.**

I. Introduction

Fuzzing is a technique developed by Barton P. Miller at the University of Wisconsin in USA. He and his colleagues have successfully used fuzzing to discover flaws in command line tools for UNIX-like systems [1], command line tools and GUI programs running under the X11 Window System [2], as well as command line tools and GUI programs running on Microsoft Windows [3] and Apple Mac OS X [4]. Using this technique, they discovered that several programs didn't handle random key presses too well, many of them crashing. Many of the problems were due to simple mistakes as neglecting to check the return value of functions before using the result. For a short introduction to fuzzing, you could read Sprundel's article from the 22nd Chaos Communication Congress [5].

While many papers have been written on fuzzing, they have mainly focused on client software on the computer, and in some cases, like Xiao et al. [6], on network protocols. What seems to be missing is research on how web applications can be tested randomly using fuzzing, and which flaws might appear. Several papers, like [7], have suggested that user input is a huge problem for web based applications, and especially with regard to command injection attacks. There are some tools available: Paros[1], SPIKE[2] and RFuzz[3] to mention some. The first two work by acting as an HTTP proxy which allows you to modify POST or GET values passed to a web site. The last one is more like a framework for fuzzing which enables a programmer to programatically fuzz web sites.

[1]http://www.parosproxy.org/
[2]http://www.immunitysec.com/resources-freesoftware.shtml
[3]http://rfuzz.rubyforge.org/

With the ubiquitous blogs and user contributed websites that exists in this Web 2.0 world, it would be interesting to find out how robust the most used applications are. When handling great amounts of user input, it is important that there is no way that input can put the web application in an undefined state, in other words: crashing it. Many programmers choose to use a web framework to avoid having to handle these problems themselves, and others make their own frameworks to simplify things.

A. Contributions

We have looked at several high profile web applications available for installation (we have not looked at hosted solutions, such as YouTube, as testing other people's production systems would be unethical), and how they handle fuzz data as input. We present a listing of flaws found in the web applications tested in Section VI, and where possible we include information on why the application failed, and how to fix the mistake, similarly as what Miller et al. did in [4]. We considered checking how these applications stand against SQL injection attacks and cross site scripting attacks, but considered this not directly related to the random testing technique we know as "fuzzing".

II. Related Work

As Miller et al. [1], [2], [4] and Forrester and Miller [3] already have stated, many applications are vulnerable to buffer overflows and similar attacks. Many of these flaws are hard for the programmer to spot, as they make the assumption that a function cannot fail and hence they do not check the returned value. Fuzzers can assist in these cases, as backed up by Oehlert [8], who found several flaws in Microsoft's HyperTerm after using a fuzzer to provide semi-valid input to the program. Microsoft's "Trustworthy Computing Security Development Lifecycle" [9] even states that "heavy emphasis on fuzz testing is a relatively recent addition to the SDL, but results to date are very encouraging."

While writing about fuzzers in [10], Stuttard and Pinto seems to expand the term fuzzer, by including other attack methods like enumeration attacks. A true fuzzer should try strictly random input, or a combination of valid and random input. Enumeration attacks might be a better approach for discovering vulnerabilities in web applications, but should not be confused with fuzzing. Stuttard and Pinto also states that analyzing results from web application vulnerability discovery is hard, and manual work is often required.

A. Client Applications

Miller et al. tested command line programs on seven different versions of UNIX [1], and managed to make upto a third of the programs hang or crash (depending on which version of UNIX they tested). When they redid the study in 1995 [2], only 9% of the programs crashed or hung on a GNU/Linux machine, while 43% of the programs had problems on a NeXT machine. Results on fuzz testing X applications (38 applications) were published in the same study, showing that 26% of the X applications crashed when tested with random legal input, and 58% crashed when given totally random input.

Bowers, Lie and Smethells redid the studies Miller et al. did on UNIX command line programs in their study from 2001 [11]. To accomodate for the fact that some of the programs originally tested had since become abandoned, they changed some of the programs for newer alternatives, i.e. replacing vim for vi. The study shows that the open source community had taken notice of Miller's study, and had indeed improved the stability of many of the affected programs.

In Forrester and Miller's study on Windows [3], 33 GUI programs were tested on Windows NT 4.0, and 14 GUI programs were tested on Windows 2000. In this study they used the API to send random messages and "random valid events". Sending random messages to the running programs caused more errors than sending valid random events. Ghosh et al. also looked at the robustness of Windows NT software using fuzzing [12]. They only tested 8 different programs, but had a lot of different test cases where they found that 23.51% of the tests resulted in a program exiting abnormally and 1.55% of the tests resulted in a program hanging.

The last study from Miller et al., conducted on Mac OS [4], shows similar results to the best results from [2] when it comes to command line programs. This comes as no surprise, as many of the command line programs in Mac OS X are GNU programs. The GUI applications on Mac OS had a worse fate. Of 30 tested programs, 22 crashed or hung, yielding a 73% failure rate.

A similar technique to fuzzing was used during the development of the Macintosh 128k which was released in 1984. The developer team created a program they called The Monkey [13] which used some APIs to send random events to the operating system. This program was a great help in the quest for bugs. Similarly there exists a program for modern UNIXes called `crashme` which has been of great help for developers of GNU/Linux in identifying rare cases where the system would crash due to erroneous input. In a whitepaper submitted to the "Black Hat USA 2007 Briefings and Training" conference [14], Miller and Honoroff outlines several useful utilities and tips for fuzzing software on Mac OS X.

B. Network Protocols and the Web

Banks et al. [15] points out that while many fuzzers exists for fuzzing network traffic, like SPIKE [16] and PROTOS [17], they don't handle stateful protocols very well, and making them do so might require more work than writing a new framework altogether. Their creation — SNOOZE — lets the user specify states and transitions for a protocol with default values for the transitions. Using this information they can write a script that creates fuzz values for some of the messages, and thus they can control which point in the protocol state machine they wish to attack, allowing them to discover bugs "hidden deep in the implementation of [the] stateful protocol."

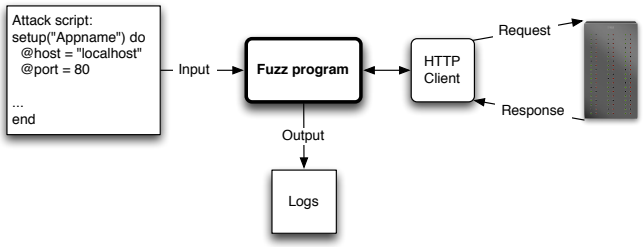

Figure 1. An overview of the main components in the fuzzer and how they interact. An attack script semi-generated by a crawler is fed to the fuzzer which in turn translates the attacks to HTTP requests which is sent to the target of the attack. The requests and their responses are then logged for manual inspection.

Fuzzing has also proven effective in discovering vulnerabilities in web browsers, and through this a means of exploiting the Apple iPhone [18]. The infamous "Month of browser bugs" article series also utilized fuzz testing in order to discover vulnerabilities in the most commonly used web browsers [19].

C. Wireless Drivers

Testing of wireless drivers are very interesting in these days, as wireless connectivity is becoming the standard for many people. It is made even more important by the fact that wireless drivers runs in kernel mode (at least on operating systems in common use), and thus an exploit can get full access to the computer, with the attacker only in proximity of the victim. Butti and Tinnès stresses this fact in their paper on discovering and exploiting wireless drivers [20], as well as the fact that the wireless networks are weakening the security perimeter.

Mendonça and Neves has done some preliminary testing of the wireless drivers in an HP iPAQ running the Windows Mobile operating system [21]. Without having the source code available, they started writing a fuzzing framework targeting the wireless drivers on the device. By running a monitor program on the device they have been able to find some weaknesses while fuzz testing the driver. Butti and Tinnès were successfull in discovering and exploiting the madwifi driver running in the GNU/Linux kernel, as well as finding several denial of service vulnerabilities in different wireless access points. Some of the findings were included in the Month of Kernel Bugs[4] project and included as modules in the Metasploit project[5].

III. Building a Fuzzer

In this section we propose a method to build a fuzzer suitable for fuzzing web applications. Our implementation

[4] http://projects.info-pull.com/mokb/
[5] http://metasploit.com/

is based on the RFuzz library for the Ruby programming language, but could just as well have been based on Peach or Sulley. An overview of how the parts are interconnected is presented in Figure 1.

In order to specify how the applications should be attacked, we have created a way of writing attack scripts for fuzzing web applications. We specify global variables for the target, like hostname and port, headers and cookies, and then we specify "attack points" for the target. The attack points in a web application are mainly web pages containing form(s) for user input.

Utilizing a random number generator, we provide convenience objects for usage in the attack scripts in the form of a "fuzz token". Each FuzzToken subclass implements a method called `fuzz`. In this method it uses the random number generator to generate random entities. The superclass also implements a `to_s` method which calls the `fuzz` method and returns the string representation of the result. Hence, the tokens are evaluated every time the HTTP client creates a request (as the request path and parameters ultimately needs to be in string format).

In the attack points we specify which path should be attacked, which (HTTP) method should be used (mainly `GET` and `POST`) and which query options should be sent. The fuzz tokens provided can be inserted as values for i.e. query options. The following listing gives an example of an attack script. The variables `word` and `fix` are fuzz tokens, and will yield a different value each time their "to string" method is called. The `word` token will give different words, the `fix` token will give different "Fixnum"s (a 30-bit signed integer), and `str(50)` gives different strings with a length of 50 characters.

```
setup "Webapp" do
  @host = "10.0.0.2"
  @port = 3000
  @headers = "HTTP_ACCEPT_CHARSET" => "utf-8,*"

  attack "search-box" do
    many :get, "/search.php",
         :query => {:q => str(50)}
    many :get, "/search.php",
         :query => {:q => fix}
  end

  attack "post-page" do
    once :get, "/login.php", :query =>
         {:user => :admin, :pass => :admin}
    many :post, "/post.php", :query =>
         {:title => word, :body => byte(50)}
  end
end
```

When the fuzzer is fed this script, it creates a Target object based on the contents. When the attack script sets a value for `@host`, it overrides the default value set by the initialization of the Target object. The `attack` method is defined to take a name and a block of code as a parameter. The code block is evaluated, and calls to `once` results in the following request getting queued once in the request queue. Calls to `many` results in the following request getting queued `@repetitions` times in the request queue. The number of repetitions is initially set to 50, but can be changed through the script.

Creating these attack scripts by hand is easy, but tedious work. We created a crawler based on Hawler[6] which outputs some generic information about the target (the `setup`-part) when it starts. Then it starts traversing the site breadth-first from the starting URI (which is given on the command line), and calls a callback function we created based on the Hpricot[7] library. The callback identifies the forms in the response, and filters out the interesting fields, creating one `attack` block per form. Finally, when the whole site is traversed, the crawler outputs the end section. By calling this script and redirecting the output to a file, we get a good starting point for writing an attack script.

We did have some problems with the crawler. While you can pass headers which it uses in each request, it is not straight forward to define pages it should abandon. This results in a problem when you add a cookie to the headers in order to "log in" to the admin panel and scrape these pages. The first couple of pages are usually parsed OK, but when it reaches the link that logs out of the admin panel, the rest of the URIs pointing within that password protected space will no longer be available.

In order to supply fuzz data as input to an application, we need to include a simple HTTP client. This client will be used to send input to the application, and return the resultant response to our fuzz program. The functionality we need from an HTTP client is the following:

1) Easy interface for creating GET and POST requests.
2) Possibility to read headers in the response.
3) Possibility to add or modify headers in the request.
4) Handling of cookies. This isn't strictly necessary, as it could be implemented through access to headers.

Lastly we have a class called Fuzzer, which is responsible for tying the components together in order to mount the attack. The Fuzzer is initialized with a Target, and creates a directory for logfiles along with a logger for the current session. Before starting the attack, the fuzz tokens found in the request queue of the target is evaluated. Recall that all we need to do to evaluate the tokens is to ask for a string representation, so mapping the to_s method on all elements in the request queue does the trick.

After evaluating the tokens, the fuzzer starts firing requests based on the information in the request queue. Using the logger, it logs requests about to be made, and the responses when they arrive. If the method used for the current request is POST, it adds the correct content type header, and puts an urlencoded version of the query in the request body, as per Section 17.13.4.1 of the HTML 4.01 specification [22]. If the method is GET, the query is passed as a part of the URI. For more on urlencoding, please refer to RFC 1738 [23], and the newer RFC 3986 [24].

[6]http://spoofed.org/files/hawler/
[7]http://code.whytheluckystiff.net/hpricot/

Having prepared the request, it uses the HTTP client to send it to the host. When the response is received, it records the status code and request timings, and logs a serialized version of the request and response. When all requests have been made, it creates one CSV file containing the recorded number of different status codes, and one CSV file containing statistics on the request timings.

IV. Using the Fuzzer

This section describes how to use the fuzzer as it stands at this point.

A. Set up Target

First, you need to set up the target you will be testing. This involves installing a web application on a test server, and will in many cases be a trivial job. Most open source applications come with a file called `INSTALL` which gives detailed install instructions. Many PHP applications also feature a web-based installer, which takes notice if it hasn't been set up properly and guides you through the steps to making it work (like i.e. Wordpress).

B. Creating the Attack Script

After setting up the application, you need to tell the fuzzer where it can send it's requests, and which parameters it can send. This can be done in many ways, but here we will describe the actions taken in this study. We did this in two steps.

First we used our crawler to crawl the web pages of the target application. The details of this has already been explained in Section III. Having crawled the site, the attack script had to be manually adjusted. The arguments to the request had to be filled in properly, as the crawler only passed the values which were suggested on the web page. As an example, consider the following: the crawler encounters a web page with a form looking like this:

```
<form action="/search" method="post">
  <input type="text" name="q" value="Search"/>
  <input type="submit" />
</form>
```

The entire document is passed to the form scraping function, but only the form is relevant here. The crawler would then output a section looking like this:

```
attack("/Welcome_to_Junebug") do
  many :post, "/search", {"q" => "Search"}
end
```

From the output we can see that on the page with URI `/Welcome_to_Junebug`, the crawler found a form that submits to the URI `/search` and which has a single input field with the name of `q` and a default value of "Search". Going through the output of the crawler later on, we might change it to something looking like this:

```
attack("Search box") do
  many :post, "/search", {:q => str(100)}
  many :post, "/search", {:q => byte(100)}
  many :post, "/search", {:q => big}
end
```

When we now choose to run the fuzzer, it will attack the search box in the following way:

1) Send "many" HTTP POST requests to `/search`, with the parameter `q` set to a random string of length 100.
2) Send "many" HTTP POST requests to `/search`, with the parameter `q` set to a random byte sequence of length 100.
3) Send "many" HTTP POST requests to `/search`, with the parameter `q` set to a random big number.

While the manual labour might sound tedious and boring (and it is), we deemed it sufficient for our initial testing. We have proposed ways to improve this part in Section VII.

C. Running the Fuzzer

Having created and tweaked the attack script, running the fuzzer is as easy as starting the application with the script as the argument: `ruby fuzz.rb targets/my_attack_script.rb`. While the fuzzer runs it will only output some information on the progress to the screen. However if you run `tail -f` on the log file, you can see a more verbose transcript of what's happening. The log file is created in `output/TARGET-NAME/`, where TARGET-NAME is the name specified after `setup`. The log file is named with the timestamp of the invocation of the fuzzer.

When the fuzzer is done, the log directory will contain the following files: A comma separated file containing the counts of various HTTP status codes (and exceptions thrown); A comma separated file containing statistics about the timings. Average, max, min times of the requests etc.; A file containing the event log; A serialized version of the requests and responses.

D. Analyzing the Results

Analyzing the results is hard to automate, since there are various ways to look at the data to determine what can be considered an erroneous response. However, we recommend starting by looking at the responses where the status code is in the 500 range. By looking at Section 10.4 of RFC 2616 [25], we see that the status codes in the 400 range are reserved for client errors which indicate that the fault is that of the client (usually the user or browser). Its Section 10.5 tells us that status codes in the 500 range are reserved for server errors, and "indicate cases in which the server is aware that it has erred or is incapable of performing the request." This is also one of the methods Stuttard and Pinto suggests using [10]. In an ideal world, we should thus be certain that if a fuzzed request resulted in a status code in the 500 range, we discovered a flaw in the application or web server. Looking at other sections we can also see that a status code 200 means success and that status codes in the 300 range is used for redirection.

We wrote a simple GUI to help filtering responses and show us the headers and response body to make this task even easier. Some of the applications will throw a stack trace at the user (maybe depending on running it in development or production

Table I
THE COMPUTERS

	Web server	Attack Machine
Brand	Cinet Smartstation 200	Apple iBook G4
CPU	Pentium III, 870 MHz	PPC G4, 1.33 GHz
RAM	377 MB	1.5 GB
Operating System	Debian GNU/Linux 4.0	Mac OS X 10.5

mode) while others will state that an error has occurred and that more is to be found in the server logs. Combining the stack trace with the source code often provides what you need to find the wrong assumptions made by the developers.

By looking at the CSV file containing counts of status codes, you should also be able to see if exceptions are raised. As an example, seeing `ErrnoECONNREFUSED` means that a connection to the web server could not be made. If this occurs after a seemingly OK request, it might mean that one of the previous requests managed to halt the web server. In that case you should check the log to find out which requests have no response. Using the CSV file you can also graph the status codes the requests generated, and using the CSV file containing timings you can plot the request timings of the runs.

V. EXPERIMENT

This section explains how we conducted our experiment. Section V-A describes the environment in which the project took place, and gives a list of computers and software used, and Section V-B gives a brief overview of the applications we tested.

A. Environment

The tests have been conducted on two machines, one web server and one attack machine (see Table I). The following software has been used on the server (the version numbers match the ones in Debian 4.0 at the time of writing): Apache 2.2.3, PHP 5.2.0-8+etch10, MySQL 5.0.32, Ruby 1.8.5 and Perl 5.8.8. On the attack machine the following software has been used: Ruby 1.8.6, RFuzz 0.9, Hawler 0.1 and Hpricot 0.6.

While testing, the machines were connected through a network cable, using an ad-hoc network with only the attacker and the server present. This way we remove the possibility of other computers interfering with our test environment, without having to set up a dedicated test lab. The server had a monitor and keyboard connected, so by running the `tail` command on the log files, we could inspect what was going on on the server while running the attack script on the attack machine.

B. Applications tested

The following is a list of the applications we have tested during the writing of this thesis. Descriptions are taken from the project pages of the respective application.

- Chyrp 1.0.3 – "a [lightweight] blogging engine, [...] driven by PHP and MySQL."
- eZ Publish 4.0.0-gpl "an Enterprise Content Management platform" using PHP and MySQL.
- Junebug 0.0.37 – "a minimalist wiki, running on Camping."
- Mephisto 0.7.3 – "a [...] web publishing system [using Ruby on Rails]."
- ozimodo 1.2.1 – "a Ruby on Rails powered tumblelog."
- Request Tracker 3.6 – "an enterprise-grade ticketing system" written in Perl.
- Sciret 1.2.0-SVN-554 – "an advanced knowledge based system" using PHP and MySQL.
- Wordpress 2.3.2 – "a state-of-the-art semantic personal publishing platform" using PHP and MySQL.

VI. FINDINGS

This section contains an overview of the discoveries we made during validation of our fuzzing tool.

a) Failure to check return values: We saw that Mephisto failed to handle an exception that was raised in a third part library used for formatting the user input, which in the earlier days of web browsers could mean that all text the user typed in was lost. A simple formatting error should be caught by the application and not result in showing the user a stack trace they usually don't understand. The programming language used, Ruby, is a dynamic language, and doesn't enforce the programmer to catch an exception or explicitly state that the exception could be thrown as in, say, Java. This might be the reason why these mistakes are easier to make in dynamic languages that enables rapid prototyping.

b) No server side validation of input: It is our belief that user data should be sanitized before being allowed to propagate through the code. You can never trust a user to enter legitimate values, even if the possible values are "limited" by a dropdown box. As we have seen it is easy to bypass these restrictions. Similarly, using JavaScript to validate user input should only be considered a convenience for the user — not a security measure. Knowing how easy it is to disable JavaScript support in a web browser, we should always enforce the same checks server side as we hope to achieve at the client side.

We found an example where Mephisto assumed that the user would not enter other values than the ones provided by a dropdown box. Failure to do so would result in an uncaught exception. While this is a bad example, it still shows that assumptions not always are correct. Also: a problem with passing user input more or less unchecked to a filter (as was done in the example mentioned earlier), is that an attacker can target a vulnerability in the third party filter in stead of the webapp itself, leading to an extended attack surface.

c) Incorrect use of HTTP status codes: While this is not really a security related bug, it is a violation of the semantics described in the HTTP protocol (RFC 2616 [25]). The biggest problem for us is that it makes automating the analysis harder, as we cannot rely on HTTP status codes to tell us how the web server and/or application perceives the error. As we stated in Section IV-D, we should, by the semantics of HTTP 1.1, be able to assert that a status code in the 500 range indicates

problems on the server. Not, as was the case with Wordpress, that the application has correctly identified that the problem originates from the user.

d) Resource exhaustion: This type of bug usually manifests itself by causing increased response times a nd possibly no response at all. This can be caused e.g. by non-terminating recursion and infinite loops. In RT, we discovered what seems to be a non-terminating recursion, resulting in high c pu consumption and a memory leak, followed by a forced process termination.

VII. Conclusion and Future Work

The tests we have been running are not comprehensive enough to give us a basis for making bold statements about the quality of the applications we have tested. However, we believe the results we found is a good indication that fuzz testing indeed can be used in combination with web applications. By running relatively few tests we managed to discover several bugs, and some potential bugs which were not investigated fully.

The biggest hurdle with fuzzing web applications is to find a good way of analyzing the results. For our purposes, checking the responses with status 500 was good enough, but for bigger result sets, other techniques might be more appropriate, like checking for certain strings in the response body (as i.e. Stuttard and Pinto does [10]).

Our work shows that some web applications indeed are vulnerable to fuzzing. Not only new and fragile applications, but also "tested and true" applications, as well as applications which has been developed with a focus on unit testing.

Proposals for future work includes:

- Using a similar approach for fuzzing web services. By parsing a WSDL file, you could automate attack script creation.
- Add "blacklisting" of pages to the crawler to avoid logging out from administrative pages.
- Make the fuzzer pick a random fuzz token for all fields with a value of "nil", and let this be the standard value generated by the crawler. This approach is similar to [20].
- Combine the crawler and fuzzer. This could make fuzzing a one-pass or two-pass job: Either crawl a page and store links, fuzz entry points on the current page, and move on; or crawl the application, log entry points and invoke the fuzzer when done crawling.
- Fuzzing file uploads might be an area worth looking into.

References

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, p. 22, Dec. 1990.

[2] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarjan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," *Computer Sciences Technical Report*, vol. 1268, p. 23, Apr. 1995.

[3] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4 WSS'00*, p. 10, Aug. 2000.

[4] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," *Proceedings of the 1st international workshop on Random testing RT '06*, p. 9, Jul. 2006.

[5] I. van Sprundel, "Fuzzing: Breaking software in an automated fashion," 22nd Chaos Communication Congress (http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf), 2005.

[6] S. Xiao, L. Deng, S. Li, and X. Wang, "Integrated tcp/ip protocol software testing for vulnerability detection," *Computer Networks and Mobile Computing, 2003. ICCNMC 2003. 2003 International Conference on*, pp. 311–319, 2003.

[7] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 2006, pp. 372–382.

[8] P. Oehlert, "Violating assumptions with fuzzing," *Security & Privacy Magazine, IEEE*, vol. 3, no. 2, pp. 58–62, 2005.

[9] S. Lipner, "The trustworthy computing security development lifecycle," in *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13.

[10] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley, 2007.

[11] B. L. Bowers, K. Lie, and G. J. Smethells, "An inquiry into the stability and reliability of unix utilities," http://pages.cs.wisc.edu/~blbowers/fuzz-2001.pdf, visited 2007-10-05. [Online]. Available: http://pages.cs.wisc.edu/~blbowers/fuzz-2001.pdf

[12] A. K. Ghosh, V. Shah, and M. Schmid, "An approach for analyzing the robustness of windows NT software," in *Proc. 21st NIST-NCSC National Information Systems Security Conference*, 1998, pp. 383–391. [Online]. Available: citeseer.ist.psu.edu/ghosh98approach.html

[13] A. Hertzfeld, *Revolution in The Valley: The Insanely Great Story of How the Mac Was Made*, 1st ed. O'Reilly Media Inc., dec 2004, pp. 184–186. [Online]. Available: http://folklore.org/StoryView.py?project=Macintosh&story=Monkey_Lives.txt

[14] C. Miller and J. Honoroff, "Hacking leopard: Tools and techniques for attacking the newest mac os x," https://www.blackhat.com/presentations/bh-usa-07/Miller/Whitepaper/bh-usa-07-miller-WP.pdf, jun 2007.

[15] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: Toward a stateful network protocol fuzzer," *Information Security*, pp. 343–358, 2006.

[16] D. Aitel, "The advantages of block-based protocol analysis for security testing," Immunity Inc., Tech. Rep., 2003.

[17] R. Kaksonen, "Software security assessment through specification mutations and fault injection," *Communications and Multimedia Security Issues of the New Century*, 2001.

[18] C. Miller, J. Honoroff, and J. Mason, "Security evaluation of apple's iphone," http://securityevaluators.com/iphone/exploitingiphone.pdf, 2007.

[19] S. Granneman, "A month of browser bugs," http://www.securityfocus.com/columnists/411, jul 2006.

[20] L. Butti and J. Tinnès, "Discovering and exploiting 802.11 wireless driver vulnerabilities," *Journal in Computer Virology*, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11416-007-0065-x

[21] M. Mendonça and N. F. Neves, "Fuzzing wi-fi drivers to locate security vulnerabilities," *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pp. 379–380, 14-16 Nov. 2007.

[22] D. Raggett, A. L. Hors, and I. Jacobs, "Html 4.01 specification," http://www.w3.org/TR/REC-html40/, dec 1999.

[23] T. Berners-Lee, L. Masinter, and M. McCahill, "Rfc 1738: Uniform resource locators (url)," http://www.ietf.org/rfc/rfc1738.txt, dec 1994.

[24] T. Berners-Lee, R. Fielding, and L. Masinter, "Rfc 3986: Uniform resource identifier (uri): Generic syntax," http://www.ietf.org/rfc/rfc3986.txt, feb 2005.

[25] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616: Hypertext transfer protocol – http/1.1," http://www.ietf.org/rfc/rfc2616.txt, jun 1999.

[26] D. Berube, *Practical Ruby Gems*. Apress, 2007, ch. Easy Text Markup with the BlueCloth Gem, pp. 45–51.