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.

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 large amounts of user input, it is important that user input can't put the web application in an undefined state, in other words: crashing it.

While some might argue that input handling and correct use of an API should be a non-issue, and a case for "secure coding practices," we'll argue that bad coders are a fact of life, and it is human to err. In an imperfect world, simple and cheap tools can aid the programmer during the implementation phase, in the hope of catching errors early. We believe this can especially benefit the fast paced web developer.

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].

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 by using semi-valid input obtained through a fuzzer. 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."

In their book about fuzzers [10], Stuttard and Pinto seems to expand the term, 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] in 1990, and managed to make up to a third of the programs hang or crash. 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.

In 2001, Bowers, Lie and Smethells [11] redid the 1990 study of Miller et al. 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, e.g. replacing vim for vi. Their study shows that the open source community had noticed Miller's study, and used it to improve 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.

B. Network Protocols and the Web

Banks et al. [13] points out that while many fuzzers exists for fuzzing network traffic, like SPIKE [14] and PROTOS [15], 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."

Fuzzing has also proven effective in discovering vulnerabilities in web browsers, and through this a means of exploiting the Apple iPhone [16]. The infamous "Month of browser bugs" article series also utilized fuzz testing in order to discover vulnerabilities in the most commonly used web browsers [17]. There are some tools available for fuzzing web applications: Paros[1], SPIKE and RFuzz[2] 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://rfuzz.rubyforge.org/

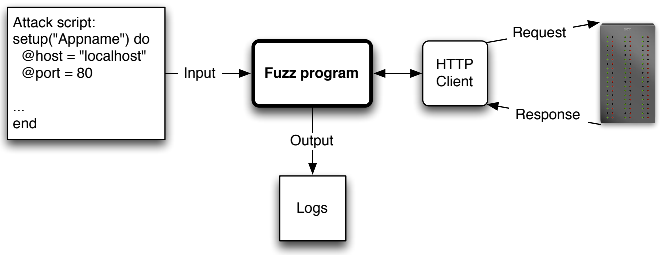

Figure 1. An overview of the main components in the fuzzer and how they interact. An attack script semi-generated by a crawler is fed to the fuzzer which in turn translates the attacks to HTTP requests which is sent to the target of the attack. The requests and their responses are then logged for manual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

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 usually runs in kernel mode, 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 [18], as well as how 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 [19]. Without having the source code available, they wrote a fuzzing framework targeting the wireless drivers on the device. By monitoring the device they have been able to find some weaknesses by fuzz testing the driver. Butti and Tinnès were successful in exploiting the madwifi driver running in the 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[3] project and included as modules in the Metasploit project[4].

III. Building the Fuzzer

In this section we propose a method to build a fuzzer suitable for fuzzing web applications. Our implementation is based on the RFuzz library for the Ruby programming language[5], 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

[3]http://projects.info-pull.com/mokb/

[4]http://metasploit.com/

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

```
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
```

Figure 2. Example of an attack script

"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 uses the `fuzz` method to get a string representation of the fuzz data. 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 e.g. query options. Figure III gives an example of an attack script. The variables `word` and `fix` are fuzz tokens, and will yield a different value each time a request is made. 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 an attack script, it creates a Target object based on the contents. When the attack script sets a value for `@host`, it overrides the default value used by 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 a predefined (and configurable) amount of times.

Creating these attack scripts by hand is easy, but tedious work. We created a crawler based on Hawler[6] which traverses the application breadth-first from the starting URI it is given. Every page is passed through a function that identifies forms, and outputs parts of the attack script. By storing the output from the crawler, 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.

[6]http://spoofed.org/files/hawler/

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 are evaluated.

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 [20]. If the method is GET, the query is passed as a part of the URI. For more on urlencoding, please refer to RFC 1738 [21], and the newer RFC 3986 [22].

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 by setting up an attack script (Section IV-A), running the fuzzer (Section IV-B) and gives hints on analyzing the results (Section IV-C).

A. *Creating the Attack Script*

After setting up the target 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 search box containing the default value "Search ...". The output would then look something 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, 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 didn't see the need to further automate it for our initial testing. We have proposed ways to improve this part in Section VII.

B. 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 attack_script.rb`. While the fuzzer runs it will only output some information on the progress to the screen. However if you monitor a log file it creates, you can see a more verbose transcript of what's happening. The log file is created in a directory based on the name specified in the attack script, and the filename is based on the time the fuzzer was invoked.

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.

C. 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 [23], 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 are used for redirection.

While looking at the logged responses with a status code of 500, some of them might contain a stack trace indicating where the application erred. In some cases, correlating the timestamp of the response with the server logs might give you the same. Combining the stack trace with the source code will often provide what you need to find out where the developer might have made an erroneous assupmtion.

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.

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

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, Section V-B gives a brief overview of the applications we tested and Section V-C briefly states how we ran the experiment with regards to the previous section.

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 monitoring the log files, we could see 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 (targets) in this study. 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.

C. Running the Experiment

After choosing targets and installing them on the web server, we generated preliminary attack scripts using the crawler, and manually tweaked them (see Section IV-A). The number of forms attacked per application, the total number of inputs (text fields, dropdown boxes, etc.) fuzzed and the time taken to manually tweak the scripts can be seen in Table II.

The time taken to tweak the scripts mainly depend on two things: how many and complex the forms are, and how much control you want over which tokens are used. eZ, Mephisto and RT all have complex forms (with many different inputs), but in the case of RT, we chose to let the fuzzer pick a random token for each input.

For Chyrp, Junebug, Mephisto and ozimodo we created one attack script for the user interface, and one for the administrative interface. For eZ, Sciret and Wordpress, we only targeted the user interface, and Request Tracker (RT) doesn't have a user interface, so there we targeted the administrative interface. For RT, we faced a problem mentioned in Section III: the crawler logged out after harvesting a few pages. We found out about this late in the process, but managed to get some results anyway.

We used the methods mentioned in Section IV-C to analyze the log files, as well as creating a chart of status codes returned, in order to get an overview of where to start looking.

VI. Findings

This section contains an overview of the discoveries we made during validation of our fuzzing tool. A list of which bugs were found in which applications is given in Table II. The issues, E1–E4, refers to the sections below.

Table II
INPUT COMPLEXITY AND BUG DISCOVERY

	Complexity			Issues			
Application	#forms	#inputs	time	E1	E2	E3	E4
Chyrp	4	11	≈15m	–	–	–	–
eZ	6	20	≈60m	–	–	–	–
Junebug	5	8	≈15m	–	–	3	–
Mephisto	10	49	≈60m	–	2	1	–
ozimodo	5	26	≈30m	–	2	–	–
RT	4	64	≈20m	1	–	–	–
Sciret	6	24	≈20m	–	–	–	–
Wordpress	4	10	≈20m	–	–	–	2
Sum	44	212	≈240m	1	4	4	2

E1 *Resource exhaustion*: 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 a non-terminating recursion, resulting in high cpu consumption and a memory leak, followed by a forced process termination. This was caused by a subroutine trying to validate our input, and after mail exchange with the developers it seems the problem revolves around bad handling of invalid UTF-8 byte sequences.

E2 *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.

E3 *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.

E4 *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 [23]). 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-C, 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.

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 effective as part of a test procedure for 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 [18].
- 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.

Acknowledgements

We would like to thank the anonymous reviewers for many useful comments.

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, (Visited May 2008).

[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 May 2008). [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] 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.

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

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

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

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

[18] 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

[19] 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.

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

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

[22] 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, (Visited May 2008).

[23] 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, (Visited May 2008).