DISCOVERY OF WEB-APPLICATION VULNERABILITIES USING FUZZING TECHNIQUES

By Michael Crouse

Dr. Errin W. Fulp, Ph.D., Advisor
The increasingly high volume of users on the web and their use of web applications has created a whole new realm of computer security. Hundreds and hundreds of new web-sites come online every day usually with some type of web-application running. The hosts of the websites and the developers of the applications use expensive code-auditing and beta testing to discover and eliminate vulnerabilities. A growing trend is the use an automated tool, or a fuzzer, to aid in testing and vulnerability discovery. Software testing tools were originally designed to test the functionality of applications; however these tool are now being used by hackers to quickly discover vulnerabilities.

This paper will discuss Web-application fuzzing, test several open source fuzzers and explore the development of a more platform independent fuzzer written in C. This paper hopes to show the effectiveness of these tools and how easily a home-brew fuzzer can be built.

**Definition 1** Definition: Fuzzing - use of automated tools to test a large a variety of input to discover undefined or insecure behavior in a program.
1 Introduction

Focus on Computer Security continues to gain momentum as more and more private information is digitized. With recent attacks against one of the worlds largest technology-based companies, Google, with large amounts of private information being stolen, the field has garnered more attention. Companies are constantly in search of new means of protection from digital attackers and with each new innovation, the pressure point is shifted.

An emerging trend is the use of pen-testers to evaluate the security of a business computer environment. Fuzzing is the process of generating and providing random, unexpected, or invalid data to inputs of a program. Fuzzing has become one of the most important tools in a pen-testers arsenal to ensure better coverage of testing. This tool allows them to test exponentially more possibilities than if they did all their testing by hand. Software companies are also putting more and more money into code auditing and security analysis teams. Fuzzing has also become a central feature of the code testing process. For example, many of the vulnerabilities in Microsoft Internet Explorer, Microsoft Word and Excel were found using fuzzing tools and techniques. Fuzzing was also the method used to discover the exploit to jailbreak the iPhone. [1]

This honor’s thesis investigates the uses of fuzzers for finding web-based vulnerabilities. Examples of vulnerabilities include directory traversal, site-mapping, injection attacks and DoS. In addition, a multi-threaded web-site fuzzer was developed to demonstrate the ease of building a custom, light-weight fuzzer.

2 All about Fuzzing

Fuzzing is a somewhat of a misnomer in that few people have a firm understanding of what exactly it entails. The first mentions of fuzzing or similar testing came from Professor Barton Miller, consider to be the father of fuzzing, and an advanced oper-
ating systems course for testing UNIX applications.[2] The course was more focused into code reliability than searching for issues with security. The roots of fuzzing come from Boundary Value Analysis or BVA but fuzzing goes to further depths than this. Fuzzing attempts to give any and all inputs that will cause an undefined behavior. It is, simply put, a brute force assault on all inputs of a system. An example would be providing all possible character combinations to a sample text field in a web-form.

2.1 Black Box Testing

Black box testing is the idea that your knowledge of a system is based only what is observable. For example, in a web-application, a user provides input, such as a login credentials, and a response, authentication or not. However, a user has no idea how the authentication process is handled, what language is implementing the application or if a database being employed. Fuzzing lends itself to this kind of testing because of how simple it is. It is an automated guess and check system that allows a person fuzzing an application or protocol to have very little knowledge of how a system works. It is also advantageous because the automated approach is easy to reproduce unlike manual testing which has a stronger human element involved. This type of testing is has the major disadvantage in that it takes a very long time to perform.

2.2 Goals of Web-Application Fuzzing

The goal of web-application fuzzing is no different than Black Box Testing, to produce an unexpected behavior. Extending this to look at web-applications specifically can be several different things. One such goal is to gain access to information outside the root web directory – the root web directory is directory where all files that are visible on the web reside. Such a vulnerability allows a hacker to learn more information such as a configuration file or retrieve a password file. Another goal is to determine whether input variables are properly handled. A similar approach can determine if
variables are passed to a language that is susceptible to a buffer overflow. Another situation that can arise when sending mangled requests to a web-server is a crash. A request or a sequence of requests that causes a system to be unresponsive could be used in a Denial-of-Service attack. Finally, performing a crawl of the website allows a hacker to build a complete site-map. This could expose improper permissions of private files and hidden executable files with more inputs to fuzz. Hidden files that take inputs can be extremely dangerous, a web-application developer might forget it exists or think it is safe because it isn’t directly accessible to the web. It could also be development code that is not yet complete or left sitting for periods of time. Fuzzing a web-application allows an hacker or tester to quickly build a list of a wide variety of possible vulnerabilities.

3 Open-Source Web-Application Fuzzers

This chapter will focus on the exploration of open-source web-application fuzzers. There are many web fuzzers that are open source and free to download. These tools are used extensively by both application developers and hackers to locate potential holes in systems. By testing and taking a closer look at a few mature fuzzing tools, one can gather information about what pieces are important to build a useful custom fuzzer. Several important details include what types of data can be fuzzed, how are requests generated, what language is used and how are requested handled. All of these questions lead to a better understanding of the process of web-application fuzzing.

3.1 SPIKE Proxy

Developed by Dave Aitel. The SPIKE PROXY fuzzer acts a proxy and accepts request through a web browser. [1] [3] A user can continue to navigate through pages and build a log of their requests. From this point, one can easily begin building tons of
test sets to start fuzzing the host. The tool allows for simple domain configuration for a range of attacks including buffer overflow, directory traversal, and SQL injection. The project is open source so code modification and enhancement is allowed. This allows a programmer to customize the fuzzer for a very specific target. SPIKE Proxy uses a web interface and is very easy to install with very little preparation. It uses OpenSSL and is written in Python so it is portable to a wide variety of systems. The tool is also multithreaded so it can generate large amounts of requests and even perform multiple tests simultaneously. Again, the multi-threaded approach is very important in dealing with HTTP requests. If the web-fuzzer was single-threaded, only one request could be created at a time and if a request is never responded to, the program would hang. When generating data that is most likely incorrect, this type of behavior would bring the process to a stand-still.

3.2 WebFuzz

Developed by Michael Sutton. This fuzzer was written as an example for a text book but is highly sufficient for basic web-application fuzzing. [1] WebFuzz is written in C# which was a good decision for several reasons: First, easy to build GUI system. Second, asynchronous socket library which allows for easy sending/receiving of network traffic. This tool is obviously not as well developed as SPIKE Proxy but has many additional features. It works completely outside of the browser, requiring the user to hand generate the requests to a degree. The GUI makes viewing the results of the audits much easier than the web interface of SPIKE Proxy. This is also open sources so making tweaks for specific tasks is fairly straightforward after spending time exploring the code, especially with the help of the textbook which highlights several key code areas. For example, it highlights how to handle multiple requests using C#’s asynchronous socket library. A multi-threaded approach is a must for building a stable web-application fuzzer.
4 C++ Fuzzer

This tool is written in entirely in C++ for the purpose of exploring the difficulty of building a custom web-application fuzzer. C++ was an appropriate language for the comfortability factor, there is a large base of programmers than are comfortable with writing C/C++ programs. The socket library is C++ is complete and reliable and writing a socket class to manage the sockets fairly simplistic. This also gives the programmer complete knowledge of how every aspect of the TCP connections are being handled from the connection to the size of the packets being sent and received. The goal was to build a generic tool that could be easily modified to add new functionalities, learn about input generation, learn more about web-application security and vulnerabilities. Finally, to gain well developed sense of what it takes to build any kind of fuzzing tool whether it be web-application or otherwise.

4.1 Generating Requests

The first major hurdle is to build a means to send a HTTP GET or POST request to a webserver. A TCP Socket classes was created that would handle socket creation, connection, and sending/receiving packets. The class leverages the C/C++ socket.h library to build all the necessary tools for network communication, more specifically, HTTP requests. By creating a TCPSocket class, it allows the actual fuzzing code to be more concise and human-readable. Overloading the output operators for send and receive is a great example of this.

Next was generating a valid HTTP GET or POST request. This required exploring the exact anatomy of a GET request. Figure x gives an example a typical request that a web-browser would make. The key fields are the first line, specifying the type of request. The common options are GET, POST, HEAD, OPTIONS, DELETE, PUT. The fuzzer only manages GET and POST but HEAD could also be a useful addition.
HEAD returns only the status code from the server response instead of the entire raw response. An initial fuzz of the site could involve only look at the status code to determine abnormal behavior that the tester could then take a closer look at. The next field is the requested file or directory and is where the majority of the work is done. GET requests include information being passed from page to page in a specific format which is where bound testing and directory traversal attempts can be made. The last field specifies which version of HTTP you are running, 1.1 has become the standard but many servers still accept 1.0 requests. The host line simply gives the name of the webserver DNS. There are tons of other pieces of information that can be included but are usually leveraged only by web-browsers to format the content. POST variables are placed in the body of the request instead of in the request path.

GET /index.html HTTP/1.1
Host: cs.wfu.edu

4.2 Handling Responses

One important part of coding a web-application fuzzer where you are going to be creating multiple connections is handling a request that does not receive a response, you do not want to stop progress or have the fuzzer to crash. If a request is made within the main thread of a program and the connection is dropped or incorrectly handled by the server, the thread would crash and the fuzz would end in the middle. The fuzzer is implements the requests with threads so the main program generates requests and then wait until the requests are responded to. The pthread library was used to create the threads and handle the wait in the main program. The thread handles the request and the response and then closes.

The response includes a header and a body, it is important to log both in order to determine if a possible vulnerability is found. It is sometimes very difficult to
determine if an expected, normal response is returned or not. It is important to pay attention to the Status Code which is displayed after the HTTP version, see Figure y. A list of status codes can be found at INSERT URL. The 400 level responses are statuses that could indicate a possible hole has been found. For instance, if a 403 is returned, the page/file exists but is password protected. Also, after a series of 404 statuses, a 500 comes back could mean the server is down, whether temporary or not. It is also important to both record the request and the response in pair and also the sequence of requests to give the best chance at being to recreate the error. I store the requests and responses into a large array structure that consists of the request, the status code, the complete response header (parsed out of the raw return) and the body of the response.

4.3 Directory Traversal

The first feature implemented for C++ Fuzzer was directory traversal. Directory traversal is the attempt to break out of the root web directory and access other files existent on the server. This is possible if the permissions on the server are incorrectly set and the URL while allowing a malicious user to reach into the system files to grab a configuration or password file. In order to fuzz this attack, design needs to be able to insert an arbitrary number of ../ into a request where the user of the package specifies. The most common places to attempt would be within a variable passed through the URL such as var in http://website.com/app.php?var=var_a. In order for a user to be able to specify where in the request they would like to attempt a traversal, the syntax [DirTraversal] within the path is used.

4.4 Buffer Overflow

Buffer overflow was once the most common security vulnerability exploited. The shift toward the web and web-applications has diminished its use. Most applications are
using languages such as Java and C-sharp that have memory management and bounds checking that all but eliminates overflows. However, many times, applications will pass off some of the work to programs written in C/C++ that is susceptible. Also, you must all consider that you are not fuzzing the applications alone but also the web server. Web servers are almost all written in languages that can be overflowed.

5 Future Work

The C++ Fuzzer is built on a sound framework that is extremely easy to expand and tailor for any specific target. Currently, the fuzzer only has the capability to fuzz directory traversal and buffer overflow. The tool should be expanded to do basic SQL injection techniques as well as cross-site scripting. Further development is also needed to allow for fuzzing of multiple variables at a time. The menu also needs to be revamped slightly for easier access to previous fuzzing.

6 Conclusion

The ease of which a tool such as C++ Fuzzer can be created demonstrates the utility of this type of tool. A custom mechanism, tailor made to audit a web application can aid in strengthening security of any web application, small or large. Web-Application Fuzzing is a topic that will continue to grow as more tools are moved online and the security of those applications become a higher priority. This project shows that with a relatively limited amount of effort and time, a more thorough audit can easily be performed to improve the overall security of these new applications.
Bibliography

[1] Fuzzing Text
