UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Using Grid Computing for Large Scale Fuzzing

Yan Xie

MESTRADO EM SEGURANÇA INFORMÁTICA

Dezembro 2010

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Using Grid Computing for Large Scale Fuzzing

Yan Xie

Orientador

Miguel Nuno Dias Alves Pupo Correia

MESTRADO EM SEGURANÇA INFORMÁTICA

Dezembro 2010

Resumo

Neste projeto, o nosso objetivo é usar a técnica de teste de fuzzing, que fornece dados inválidos, inesperados ou aleatórios para a entrada de um programa para nele tentar encontrar vulnerabilidades. Os resultados do teste fornecem ao programador informações para melhorar o programa, nomedamente para torná-lo mais seguro. Um ambiente de computação em grade é usado para suportar o fuzzing das aplicações usando simultaneamente os recursos de vários computadores em uma rede, a fim de paralelizar o processo e permitir tentar muitas entradas diferentes. Um trabalho de fuzzing é dividido em várias tarefas de fuzzing e distribuído aos recursos de rede que se encontrem livres para que a operação seja realizada. Um broker recebe as solicitações de fuzzing de clientes, e insere a divisão de tarefas num servidor Web, como o Apache. Quando os recursos da rede estão disponíveis, as tarefas de difusão são descarregadas a partir do servidor web e automaticamente executadas e os resultados retornados ao serviço de coordenação. O serviço de coordenação Zookeeper é usado para sincronizar o broker, o servidor web e dos recursos.

Palavras-chave: Fuzzing, computação em grade, ZooKeeper, testes de segurança

Abstract

In this project, our goal is to use a testing technique called fuzzing that provides invalid, unexpected or random data to the input fields of an application to find vulnerabilities in the same application. The testing results provide a programmer with information to improve the program, making it more secure. A Grid computing environment was designed to support the fuzzing of applications, by using simultaneously the resources of many computers in a network, in order to parallelize the process and allow trying many different inputs. One fuzzing job is divided into many fuzzing tasks and distributed to the free network resources for fuzzing. A broker gets the fuzzing requests from clients, and then inserts the split fuzzing tasks into a Web server, like Apache. When resources in the network are available, fuzzing tasks will be downloaded from the web server and resources will automatically execute these tasks and return the results to ZooKeeper. The ZooKeeper coordination service is used for synchronizing the broker, the web server and the resources.

Key words: fuzzing, grid computing, ZooKeeper, security testing

Acknowledgments

I sincerely thank the professors in the CMU/FCUL master program, for imparting valuable knowledge to me, and I would like to give special thanks to my thesis advisor Miguel Pupo Correia. I also thank all my colleagues in the master program. You guys gave me lots of help during the program, and I wish you have a prosperous career and an enjoyable life.

Yan Xie
December 2010

Dedicated to my family and all my friends

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Software security, which is worth being paid much more attention to than ever before, is currently a serious problem around the world. As the growing reliance on internet services makes malicious intrusions more attractive, attacks on systems become more and more prominent. Examples are cross-site scripting (XSS) [15], SQL injection [30] and so on. Consequently, numerous innocent internet users have suffered tremendous losses from system failures or being trapped in attacker's tricks, such as identity theft [47] and password leakage. If such trends continue and expand without any effective intervention, the problem will become worse in the future.

Confronted with this situation, enterprises and governments expect more secure techniques which will help develop software in a more secured manner or of improved software security. Unfortunately, common software development practices inevitably leave software with vulnerabilities, because of the fact that most software developers focus more on function implementations, rather on vulnerability prevention [1]. Even in software that was carefully designed by considering security, vulnerabilities can always find places in those complicated-designed systems. As a result, what developers count on is to perform software testing [31-33] after the system is developed.

Traditionally, testing software refers mainly to requirement-based function testing [34] to verify a specific action or function of the code, which is intended to answer "can user do this or that". It can help to detect inconsistency and incompleteness in the system. On the contrary, in non-functional testing [35, 37] the objective is to test those requirements that do not relate with functionalities, concerning more on system security, performance, reliability, etc. Absence of non-functional testing buries invisible problems in the system, which eventually leads to system malfunction if some unexpected operation or ill-intended action is done. Digging out these problems in the system requires persons with adequate skills and tools.

In this thesis, we will only focus on one aspect of non-functional testing—security testing [6, 7, 36]. For detecting and getting rid of vulnerabilities, security testing on software is considered as a good measure due to the following reasons. Firstly, no matter how well today´s systems have been developed, they are often complicated with huge volumes of code, complex internal interactions, interoperability with uncertain external components, unknown interdependencies coupled with vendor costs and schedule pressures, which means that exploitable flaws will always be present or surface over time. Thus security testing plays a crucial role in filling the gap between the state-of-the-art in system development and actual operation of these systems. Secondly, security testing is so important to understand, calibrate, and document the operational security posture of an organization, that it is an essential

component to improve the security of organizations. Therefore, organizations that have an organized, systematic, comprehensive, ongoing, and priority driven security testing regimen are in a much better position to make prudent investments to enhance the security of their systems.

1.1 Problem statement

To perform security testing of large applications, we consider the following scenarios: in companies or governments, they use software in many fields, such as providing services to the public. Generally, these companies or governments demand complicated software to satisfy their needs. As a result, software has to be designed, and developed in a complex manner. Due to this situation, software vulnerabilities inevitably exist, which might allow attacks on the system in the future. For eliminating these vulnerabilities and preventing giant losses suffered from attacks, they have to perform security testing on the system. Alternatively, they can send their application to security testing agencies for obtaining vulnerability details if they prefer to obviate the boring testing process or they do not have enough hardware resources or software support to perform security testing.

Independent of the security testing being performed by software development companies or security testing agencies, they can not escape the difficult situation of local resource limitation if they plan to test software in a centralized manner. The problem becomes much more prominent for security testing agencies, because they have to promise to perform security testing in a short period at the request of their clients, while they receive too many security testing jobs from different enterprises or governments. Due to budget and space limitation for resources in a local environment (such as CPUs, memories, hard disks, etc.), they can not test software efficiently to satisfy all clients' demands if given too many applications for security testing in a short period.

1.2 Solution

To address the problem of resource limitation, in this thesis we propose a grid computing environment for fuzzing, which uses free network resources to perform security testing. When performing a security test, one fuzzing job will be divided into many fuzzing tasks. We intend to use the fuzzing tool SPIKE [9] to help create fuzzing tasks. The grid computing environment distributes these fuzzing tasks to the remote resources which committed to perform security testing. For example, one fuzzing task fuzzes the field 'Name', one fuzzes "Contact", and another fuzzing task takes charge of the field 'Address'. The three fuzzing tasks are distributed in the network, and ideally remote resources can fuzz the three fields respectively at the same time. Thus fuzzing process can go on efficiently by reducing almost two-thirds of fuzzing time spent in a local fuzzing environment.

划分任务工具：SPIKE
下面有划分例子

ZooKeeper is used to coordinate the fuzzing service of remote resources, from which they fetch fuzzing tasks referred by task znodes [12] in ZooKeeper. The broker publishes invitations on the website (e.g., a link directed to the fuzzing process).Once network resources agree to join the fuzzing process (e.g., clicking the invitation link), they will be automatically directed to download fuzzing tasks and execute them in the remote resources' local environment. Execution results from remote resources are available in ZooKeeper's task znodes, which will be taken away by the broker who is responsible for delivering details of vulnerabilities to companies or governments.

By designing the fuzzing system in a distributed manner, thousands of remote resources can be invited to join the fuzzing service, eliminating inefficiency on the security testing. Besides all the functionalities introduced, we also consider achieving high availability and performance in the system by accomplishing some important attributes, such as preventing concurrence [60]. 保持availability需要解决竞争问题，看[60]

The outline of this thesis is arranged as follows. Some related work will be discussed in the following sections. In Chapter 2, we are going to represent some basic concepts in security, some attacks related with input validation, and techniques that will be implemented in the system. Then we introduce the development environment of this system. In Chapter 3, the system model will be presented. System components and their functionalities are explained in detail. Chapter 4 will provide more technical details about how the system is constructed. In Chapter 5, we intend to perform fuzzing testing on three programs, and some strategies will be presented to give hints on how to fuzz programs and improve programs with fuzzing results. At the end of the thesis, we make some conclusions and propose some interesting issues in the prototype of the grid fuzzing system.

1.3 Related work

Regarding the grid environment, we borrow the idea from a GRIDTS [5]. In GRIDTS, one job is divided into many tasks, and a tuple space is used for supporting task scheduling. Tasks to be executed are placed in the tuple space. The grid resources retrieve unexecuted tasks from the tuple space and execute them. Results are also placed in the tuple space, available for users. What mainly differentiates our grid fuzzing system from GRIDTS is that we use ZooKeeper to coordinate the service, rather than the tuple space. Also, we consider a specific usage for the grid fuzzing, while GRIDTS is a general purpose grid infrastructure. There are many other grid environments, such as Globus [29] and OurGrid [38], but they are not based on coordination services like a tuple space or ZooKeeper.

The idea of attack injection to find vulnerabilities is introduced in AJECT [2]. The paper describes that the tool AJECT generates attacks with respect to some pre-defined test classes by using a specification of server's communication protocol.

AJECT performs these attacks through the network while it monitors the behavior of the server both from a client perspective and inside the target machine. Incorrect behavior indicates a successful attack and potential existence of vulnerability. A fuzzing system performs a brute force attack without pre-defining test classes [17]. It will not monitor state of the target, and the system will keep on fuzzing until it crashes. By obtaining those crashes in the system, vulnerabilities can be identified.

There is no publicly available similar work or study concerned with distributed fuzzing system, although some premature concepts have been allegedly used by Microsoft to find 1800 Office bugs [17]. There are several fuzzing frameworks and tools, such as SPIKE [9], Powerfuzzer [48], OWASP JBroFuzz [49], etc. These 几个分布式框架 frameworks are implemented in a centralized manner, so that they can not fuzz applications in a complete way due to limitations of time and resources. To improve these fuzzing frameworks, we take advantage of a grid environment to design a distributed and extensible fuzzing system, in which we use SPIKE as an example framework to perform fuzzing. All fuzzing tasks are independent, and are distributed to the network resources. With aids from remote resources, the fuzzing system achieves the goal of high efficiency in performing fuzzing applications.

For the purpose of coordinating service, an alternative to ZooKeeper is DepSpace [10]. As the system considers an unlimited set of clients that interact with a set of n servers, design and implementation of secure and fault-tolerant tuple space in n servers are investigated in DepSpace. The tuple space implemented by a set of tuple space servers can be considered as a shared memory object which provides operations for storing and retrieving ordered data sets. As long as less than a third of service replicas are assumed to be faulty, the service offered by DepSpace is secure, reliable and available. In this project, we use ZooKeeper [12, 24] to coordinate distributed services. ZooKeeper incorporates elements from group messaging, shared registers, and distributes lock services in replicated and centralized services. ZooKeeper provides a platform for distributed processes to coordinate with each other through a shared hierarchal name space of data registers (znodes), much like a file system. It provides the abstraction of znodes that can be manipulated through the ZooKeeper API. To avoid complicated situation of processing requests which depend on responses and failure detection of other clients, an API is intended to manipulate simple wait-free data objects organized hierarchically. Both DepSpace and ZooKeeper use replication, yet there is important difference. For DepSpace, replicas are supposed to tolerate Byzantine failures; while ZooKeeper's replicated service is mainly to tolerate replicas' crashes, achieving high availability and performance.

Chapter 2 Concepts and Techniques

There are two basic concepts in security: trustworthiness and trust. Trustworthiness measures how much a component, subsystem or system meets a set of properties. Trust defines the accepted dependence of a component on a set of properties of another component, subsystem or system. These properties refer to functional or non-functional. From perspectives of trustworthiness and trust, trust can be put on an untrustworthy system even though the system does not satisfy a set of properties. Specifically, both software and its users trust much stuff which should not be trusted. For example, software bought from other companies or other countries may have a backdoor left by the programmer. As a result, it is high likely that the users will suffer losses from the untrusted system. So it is better to evaluate the system's trustworthiness before placing trust on system components.

By definition, vulnerabilities are system defects that may be exploited by an attacker to subvert the security policies, thereby impact system's confidentiality, integrity or availability. Vulnerabilities are exactly those untrustworthy elements which should be eliminated in the system. Basically, there are three types of vulnerabilities: design vulnerabilities, implementation vulnerabilities, and operational vulnerabilities. They can be exploited through attack interfaces [16], which are collections of possible entry points accessed by anyone regardless of their roles in the system. Specifically, attack surface refers to pre-defined components by software developers when the system is in the design phase, with which anyone can interact, such as socket and inter-process communication [50], APIs, files, user interface, operating system, environment variables and program arguments, etc. Malicious attackers can take advantage of entry points to explore vulnerabilities in a system. Consequently, a successful intrusion is achieved by attacking on vulnerabilities.

As mentioned before, theoretically trust should not be placed on those untrustworthy systems. In practice, it is the reverse. As a result, malformed inputs can be taken advantage of by an attacker to perform attack injection in a system if some vulnerability is successfully explored. The inputs can never be trusted in terms of the following aspects: first, an attacker can pass malformed arguments to any program parameter. For example, even though the shell imposes limits on input, an attacker may still be able to call the program by getting around the shell. Second, things left by the parent process can be taken advantage of by an attacker. Also, environment variables can result in an attacker gaining root access, giving the attacker more privileges to do something unexpected to the system, such as deleting system files, modifying system configurations, etc.

This chapter is composed of three sections. First, we introduce three types of attacks. Then the main techniques and concepts used in this thesis are presented: grid computing, fuzzing, ZooKeeper, Apache. The third section briefly describes the

development environment used for this thesis.

2.1 Attacks

Considering that the objective of a fuzzing tool SPIKE's capability to find vulnerabilities, we present three vulnerabilities and the corresponding attacks in this section: buffer overflow, SQL injection and format string. These three vulnerabilities are some of the most often discovered using SPIKE.

2.1.1 Buffer overflow

Buffer overflows [57, 64] happen when use of a buffer is not checked, such as neglecting to check data size, which allows an attacker to overwrite data that controls the program execution path and hijack the control of the program to execute the attacker's code instead of program code. As data is mainly stored in stack, heap, or BSS(block started by symbol) [59] segments, the overflowed buffer will cause the application that owns the buffer to become unstable, or crash, resulting in denial of service (DoS) [58]. Programs written in C/C++ language are most susceptible to buffer overflow attacks.

There are mainly two types of overflow attacks: stack overflow [61] and heap overflow [62, 63]. Stack overflow occurs when too much memory is used on the call stack. The call stack has limited amount of memory, which is determined at the start of one program. Its size depends on many aspects, such as machine architecture, programing language, multi-threading, etc. When too much memory is used on the stack, it is highly likely to result in a program crash. Heap overflow happens in the heap data area. As memory on the heap is dynamically allocated by the program at run-time, the attack is launched by corrupting data in the memory to cause the application to overwrite internal structures, such as linked list structures. As a result, data at specific location can be altered in an arbitrary way, or arbitrary code can be executed.

2.1.2 SQL injection

To understand how the input is ill-constructed to incur security problems, we would like to introduce the concept of metacharacter. A metacharacter [51] refers to a special character in a program or data field which provides information about other characters, such as '^', '*', '|', ';', etc. Rather than their special meaning, these metacharacters can be used by attackers to explore vulnerabilities in the system. The most typical problems are embedded delimiters, NULL character injection and truncation.

In applications, the input fields always trust commands given by users containing only characters, not metacharacters. As a result, metacharacters are always introduced by attacker to explore vulnerabilities, which appear when constructing strings with filenames, registry paths (Microsoft Windows), Email addresses, SQL statements and adding user data to file.

SQL injection is the most serious and wide-spread threat based upon lack of proper input validation. From the OWASP Top 10 2010 [22], SQL injection is in the first position of the top 10. SQL injection is a code-injection attack where data submitted by the user is included in an SQL query, such that part of user's data is treated as SQL code [30]. Database often contains sensitive information, such as user account number, password, etc. As a result, database security violations can cause identity theft, loss of confidential information. The following lists the strategies frequently used in SQL injection:

- Tautology: inject code in one or more conditional statements so that they always evaluate to true. The intent is to bypass authentication, identify injectable parameters and extract data.

- Union query: exploits a vulnerable parameter to change the data set returned for a given query, so that application can be tricked to return data from a table which is not intended by the developer. The intent is to bypass authentication to extract data.

- Piggy-backed query: add additional queries into the original query, and require database configured to accept multiple statements in a single string. The intent is to extract data, add or modify data, perform denial of service attack, and execute remote commands.

- Stored procedures: execute stored procedures present in the database, statements can be passed to stored procedures, vulnerable to piggy-backed query, union query. The intent is to escalate privilege, perform denial of service and execute remote commands.

- Illegal/incorrect queries: inject statements that cause syntax error, type error, and logical error. Syntax error can be used to identify injectable parameters, and type error and logical error can deduce data types and reveal names of tables and columns that caused error. The intent is to identify injectable parameters, perform database finger-printing, and extract data.

- Inference: modify the query to recast it in the form of an action that is executed based upon the answer to a true or false question. Blind injection (infer information by asking true or false) and timing attacks (gain information by observing time delays from response) are the two well-known attack techniques.

The intent is to identify injectable parameters, extract data, and determine data schema.

- Alternate encodings: modify the injected text so as to avoid detections of defensive coding practice and automated prevention techniques (e.g., Hexadecimal, ASCII, and Unicode). Common scanning and detection techniques do not to evaluate all specially encoded strings, leaving these attacks to go undetected. The intent of alternate encoding is to evade detection.

2.1.3 Format string

Format string vulnerabilities [64] happen most frequently when a programmer prints a string which contains data supplied by user. The format string attack stems from the use of unfiltered user inputs which performs formatting, such as mistakenly writing printf(buffer). For example, the format token "%s" can be used to print data from the stack or other locations in memory. Also, an attacker can write arbitrary data to any locations by using "%n" format token. A typical exploit is to combine these techniques to force a program to overwrite the address of a library function or the return address in the stack with a pointer pointing to malicious shell code. Most of the format string bugs are caused by C language's non-type-safe argument passing conventions.

2.2 Techniques

As described in section 1.1, large applications demand much more resources for security testing. We are going to discuss in detail how to establish the fuzzing system to test large application's vulnerabilities efficiently. In this section, several important techniques used in this project are listed.

2.2.1 Grid computing

In grid computing [4, 18, 23, 41] (or in a computational grid), the solution of a scientific or technical problem, which usually requires a great number of computer processing cycles or access to a large amount of data, is accomplished by using the remote resources of many computers in the network. Grid computing requires the use of software that can divide and farm out pieces of a program to as many as thousands of computers in the network. It can be regarded as distributed and large-scale cluster computing [52] and as a form of network-distributed parallel processing. Grid computing can be confined to the network of computer workstations within a corporation or it can be a public collaboration. A well-known example of grid computing in the public domain is the ongoing SETI [39, 40] (Search for Extraterrestrial Intelligence) @Home project, in which thousands of people share

the unused processor cycles of their PCs in the vast search for signs of "rational" signals from outer space. Figure 2.1 represents the architecture of a grid computing system.

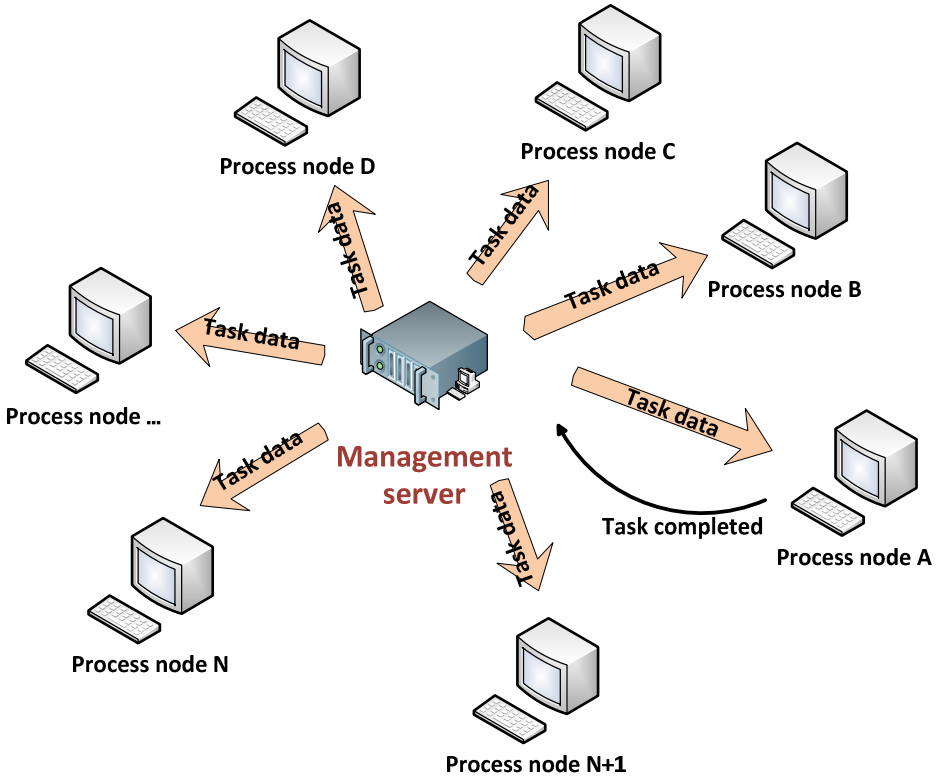


Figure 2.1 Grid Computing Environment

Grid computing coordinates disparate information technology resources across the network using middleware, which allows them to function as a whole virtually. In computational grids, a wide variety of geographically distributed computational resources, such as supercomputers, computer clusters, storage systems, data sources, instruments, people, can share, select and aggregate information; and they can be presented as single or unified resources for solving large-scale and data-intensive computing applications (e.g., molecular modeling for drug design brain activity analysis, and high energy physics). A grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery, and resource access.

Grid computing uses a given amount of computer resources more cost-effectively, solving problems that can not be approached without an enormous amount of computing power. The computing grid's goal, like that of the electrical grid, is to provide users with access to the resources whenever they need them. Grids have realized two goals: providing remote access to information technology assets, and aggregating processing power. The most obvious resource included in a grid is a processor, but grids also encompass sensors, data-storage systems, applications and

other resources.

2.2.2 Fuzzing

As mentioned in Chapter 1, traditional testing focuses mostly on verifying functional properties, so that the testers build test cases and scenarios based on the system requirements. For finding vulnerabilities, like command injection [53], this approach does not work. What traditional testing missed are things that the software should not do or should not allow, and that is strongly required in security testing.

Security testing includes many aspects. In this thesis, we studied performing security testing in the input space. Theoretically, all possible combination of inputs should be sent to the application for exploring vulnerabilities in system. For example, considering the case of 10 forms/application, 10 fields/form, and 62 characters/field, the ideal test inputs is: $10 \times 10 \times 62^{50}$. The example just counts how many valid inputs there are to test those fields. If considering invalid characters, such as ' ', '<', '%', '|', ';', '*', more inputs have to be tested. Furthermore, there are much more input fields in complicated systems. Thus, it can be concluded that much more time and efforts have to be spent on those complicated systems for security testing. And the fact also indicates that security testing requires more CPUs, memories, hard disk storage than functional based testing, etc.

Fuzzing [9, 43, 44] is a software testing technique, which provides invalid, unexpected or random data to the inputs of a program. If the program fails (e.g., by crashing), the defects can be noted, which can be of help to improve software quality. There are three kinds of fuzzing:

- Random fuzzing: random inputs are generated;
- Recursive fuzzing: iterating through all combination of characters from an alphabet;
- Replacive fuzzing: iterating through a set of predefined values – fuzz vectors.

Ideally, all type of program inputs can be fuzzed, but file formats and network protocols are the most common targets of fuzzing. Those interesting inputs include environment variables, keyboard and mouse events, and sequences of API calls. Even the items which are not normally considered "input" can also be fuzzed, such as the contents of databases, shared memories, or the precise interleaving of threads [43]. Inputs may cross trust boundaries [54], such as network sockets, pipes, RPC interfaces. These input fields strongly require fuzzing for addressing security issues.

Fuzzing is often used in large software development projects that employ black-box testing [45]. However, fuzzing is neither substitute nor formal method for exhaustive testing, because it can only provide a random sample of the system's behavior, not an overview of the system security. In many cases passing a fuzzing may only

demonstrate that a piece of software can handle exceptions without crashing, rather than behaving correctly. Thus, fuzzing can only be regarded as an assurance of overall quality rather than a bug-finding tool. As a gross measurement of reliability, fuzzing can suggest which parts of a program should get special attention, in the form of a code audit, application of static analysis [55], or partially programming applications.

Fuzzing is accomplished with the help of a fuzzer. At a high level, most fuzzers can be categorized into four groups [56]: file fuzzer (file formats), network fuzzer (network protocols), general fuzzer (file, network, custom I/O interfaces), and customer/one-off fuzzer (a specific format or network protocol). Fuzzing can be broken up into six phases:

- Investigate: determine what to fuzz;
- Modeling: model data and state of the target system;
- Validate: verify the model is correct and the fuzzer can indeed talk to the system in a meaningful way;
- Monitor: make sure the target system can be monitored;
- Run: run the fuzzer;
- Review results: review the findings.

As mentioned in section 1.3, there are many fuzzing frameworks, such as Powerfuzzer, OWASP JBroFuzz, Peach Fuzzing Platform, SPIKE, and Evolutionary Fuzzing System. In our system, we use SPIKE framework.

SPIKE is a fuzzer creator which has pre-defined fuzzers, and new fuzzers can be added as well. As long as most protocols are built around similar data formatting primitives, SPIKE supports testing in most of these data formats. The SPIKE can quickly reproduce a complex binary protocol, and easily mess with protocols.

Specifically, SPIKE uses data structures, which support lengths and blocks. For testing applications, programmers have to define a new spike that will be modified and injected. The input of the program must be understood first. And then fields must be identified in the input that will be fuzzed. By taking advantage of the loop support in SPIKE, we can iterate through all possible combinations to see if they cause any aberrant behavior. The spike must be set and initialized before fuzzing. As the SPIKE is a kind of “First In First Out” queue, the content in spike must be cleared before each round of combination is sent to the buffer. If there is large data payload in SPIKE, script can be used to play quickly with the protocol, which can be parsed to call any functions found within.

2.2.3 ZooKeeper coordination service

ZooKeeper is a centralized service for maintaining configuration, naming, providing distributed synchronization and group services. All of these kinds of services are used

in some form or another by distributed applications. ZooKeeper itself is intended to be replicated over a set of hosts, as shown in Figure 2.2 [8, 12].

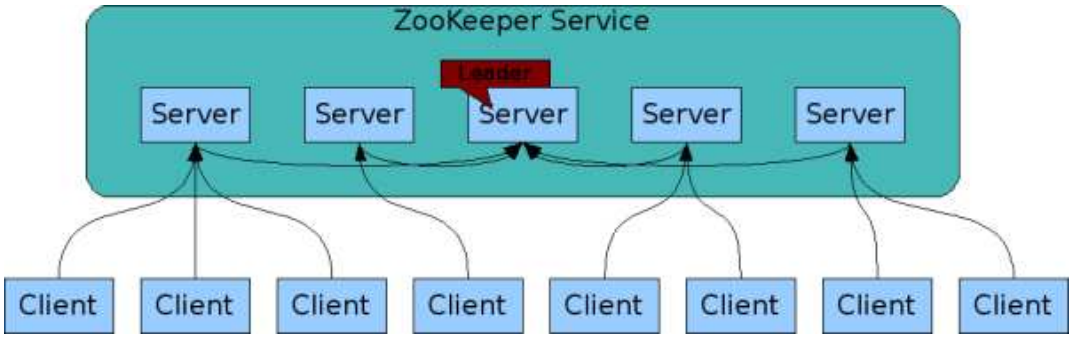


Figure 2.2 ZooKeeper [24]

ZooKeeper coordinates distributed processes through a shared hierarchal namespace which is organized like a standard file system. To represent the hierarchal namespace, the ZooKeeper use znodes, similar to directories and files. There are two types of znode states: regular and ephemeral. The regular znode is established and deleted explicitly, and it can have children. The ephemeral node can be deleted explicitly or removed automatically when the session terminates. Children are not allowed in ephemeral node. Typically, a file system is intended for storage, but ZooKeeper is not. The ZooKeeper's data is kept in memory, that data information can be obtained timely, achieving high throughput and low latency [12].

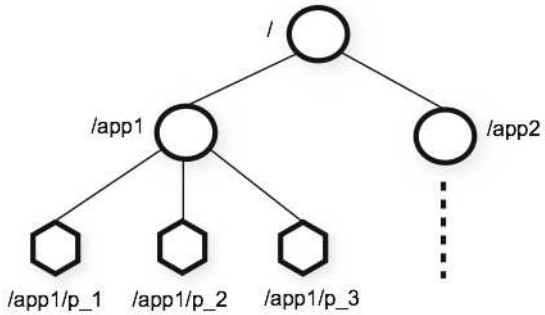


Figure 2.3 ZooKeeper's Hierarchical Namespace [24]

ZooKeeper's goal is to extract the essence of different distributed processes into an easy-communicable interface for coordinating their services. The ZooKeeper implements consensus, group management, and presence protocols, which saves efforts of distributed processes to apply them [12].

2.2.4 Apache HTTP server

The Apache HTTP Server [14] is a robust, commercial-grade, and freely-available

source code implementation of an HTTP (Web) server, which is notable for playing a key role in the initial growth of the World Wide Web (WWW). Most web servers using Apache run a Unix-like operating system, which behave similarly to a UNIX system, but not necessarily conform to or are certified to any version of UNIX specification. Apache is primarily used to provide service for both static content and dynamic web pages on the World Wide Web [25]. In our system, as ZooKeeper is not designed for storage purpose, Apache HTTP server is mainly used for storing files, paralleling with ZooKeeper's records. The Apache HTTP server provides a downloading service to remote resources, where files need to be available in a secure and reliable way. In addition, the HTTP server can provide download service.

2.3 Development environment

Development of the project was based on a local-area environment composed by VMware virtual machines [19], in which the Apache HTTP server, the resource, the broker, and ZooKeeper can be executed.

Overall speaking, the Apache HTTP server was installed in the local environment, and the broker split a fuzzing job into many tasks and inserted these fuzzing tasks into the Apache HTTP server, while established corresponding znodes in ZooKeeper. A local resource is developed to testing whether fuzzing tasks could be downloaded and executed.

Later, the grid fuzzing system was tested in a distributed manner. One computer was used as the server to install the ZooKeeper server and HTTP server, and the computer was deemed as the grid fuzzing server providing fuzzing services in the network. Several other computers acted as remote resources, which were directed to connect the fuzzing server to contribute fuzzing applications.

The environment used involved the following:

- OS: Ubuntu 9.10, Windows XP
- Tools: Apache HTTP Server 2.2, ZooKeeper 3.3.2, VMware 7.0, Eclipse 3.5.2, Java 1.6.0_15, SPIKE
- Languages: Java, C

Chapter 3 The Grid Fuzzing System

The goal of the project is to simplify the complicated, time-consuming and resource-consuming process of software security testing. For attaining this goal, we propose a grid fuzzing system to allow fuzzing applications using many free resources in the network, in order to parallelize the fuzzing process and allow trying many different inputs. There are some initial concepts that have to be kept in mind: a fuzzing job refers to the application (from the client) the grid fuzzing system fuzzes; a fuzzing task refers to part of the application input space that are fuzzed; one fuzzing job is composed of many fuzzing tasks, and thus fuzzing all tasks in one fuzzing job successfully represents one fuzzing job is finished.

For coordinating those remote resources to fetch tasks, fuzz tasks and upload fuzzing results, we take advantage of the ZooKeeper coordination service and the Apache HTTP server. Specifically, divided fuzzing tasks will be stored in the HTTP server, while information of tasks will be recorded in ZooKeeper. The grid environment system will publish some links in order to invite remote resources' participation. Once a remote resource agrees to contribute the fuzzing process, it will go to the ZooKeeper and pick the description of a task, then to the Apache HTTP server, and download the task, and finally perform the fuzzing. The record corresponding to the fuzzing task state in ZooKeeper is modified by remote resource which fuzzed the task, in order to prevent other resources from getting the same fuzzing task. By making use of free resources in the network to fuzz tasks, fuzzing one application can be accomplished efficiently. At the same time, free resources in the network are utilized in a much more efficient way.

As fuzzing jobs is the main objective of the grid fuzzing system, the core of the fuzzing system is to coordinate the broker, HTTP server, Zookeepr and remote resources to achieve fuzzing all fuzzing tasks of each fuzzing job. Figure 3.1 demonstrates how the system works.

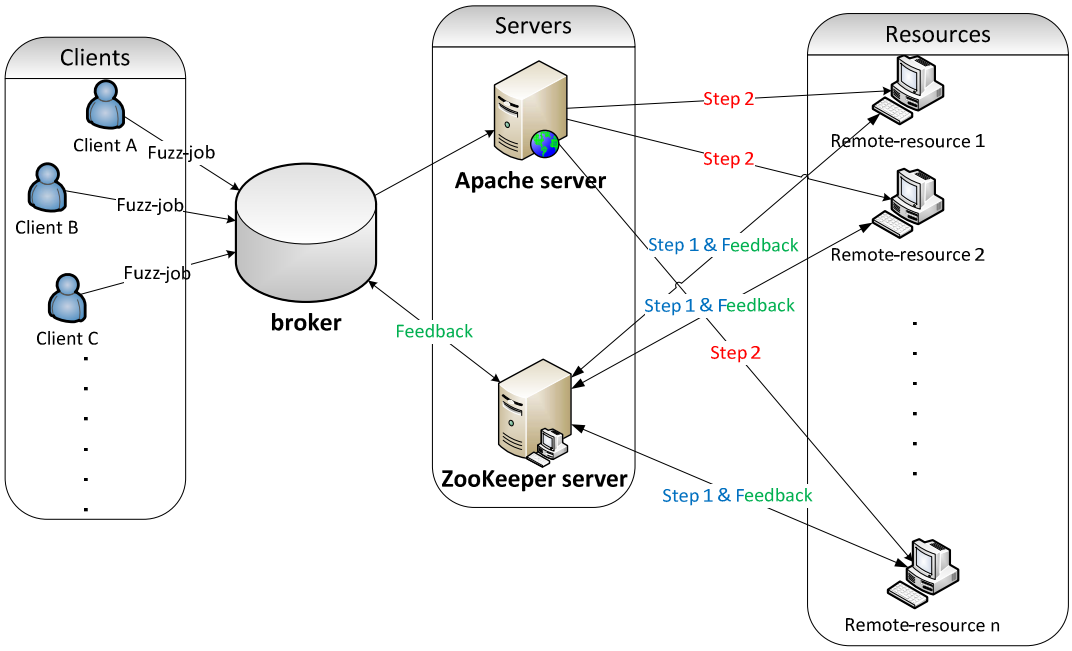


Figure 3.1 Schematic architecture of the grid fuzzing system

The meanings of the labels used in the figure are explained in the following:

- Client A(B,C…)—those entities who want to fuzz their applications;
- Broker—responsible for splitting a fuzzing job into many fuzzing tasks;
- Apache HTTP Server—provides storing and downloading of fuzzing tasks;
- ZooKeeper Server—parallels with Apache HTTP server for coordinating service;
- Step 1—get unexecuted fuzzing task information from ZooKeeper server;
- Step 2—based on the ZooKeeper information, fetch (download) corresponding fuzzing task from Apache server ;
- Feedback—execution results of fuzzing tasks which will be finally sent back to clients.

3.1 System Components

The grid fuzzing system includes the components of clients, the broker, ZooKeeper server, Apache HTTP server and remote resources. Next we explain their roles played in the system in more detail.

3.1.1 Clients

Clients are those entities who want to fuzz their applications, such as governments, financial departments. Usually, they concentrate more on using applications for daily business or management issues, so that they prefer to obviate the boring process of testing applications, or they do not have professional staff, enough hardware or software resources to support security testing. Nevertheless, to ensure that their

applications are secure enough, they send applications to the broker and possibly pay for the security testing. Clients will be informed of feedback by the broker when security testing results are available.

3.1.2 Broker

Broker's role is played by entities like a software-development company or security testing agency. Generally, the broker has two responsibilities: one is coordinating fuzzing service, and the other is fuzzing job and task states management (e.g., task execution failure).

As fuzzing one application requires a large amount of time and resources, the broker is mainly responsible to coordinate fuzzing tasks in remote resources using ZooKeeper and the HTTP server. The broker first splits one fuzzing job into many fuzzing tasks, and then uploads split fuzzing tasks to the corresponding pre-established fuzzing job directory in the Apache HTTP server, while records the fuzzing job and tasks information in ZooKeeper's znodes.

Besides coordinating service, we also consider other features of the broker. It is inevitable that occasionally something unexpected occurs on the remote resource side, such as downloading failure, fuzzing failure. To prevent such accidents, each remote resource will be given a fixed amount of time for downloading, executing a fuzzing task and updating the corresponding znode with the fuzzing results. What the broker does is to monitor each task periodically, checking if a fuzzing task is out of time. If so, the task state will be reset in the ZooKeeper, so that other remote resources can compete to fetch the task as usual.

The broker also updates fuzzing job information by checking if all fuzzing tasks are finished. Once the broker notices one fuzzing job's information is changed, which indicates that the job is finished, the job state will be replaced with a new state. For collecting fuzzing results of one fuzzing job, the broker would check each fuzzing job node at a certain time. If there is change in the job node state, the broker will immediately collect all fuzzing results from the fuzzing task znodes under the fuzzing job znode in ZooKeeper.

After fuzzing results collection is done, the broker will delete the job node and all task nodes in ZooKeeper, as well delete fuzzing task files and fuzzing job directory in the HTTP server. The fuzzing results will be kept in a .log file, which will be sent back to the clients.

3.1.3 Servers

There are two servers worked in the grid fuzzing system—Apache HTTP server and

ZooKeeper server. The HTTP server is mainly used by the broker to establish fuzzing job directories for storing fuzzing tasks, and provides downloading fuzzing tasks (executable binary file developed by C language) to remote resources in the network.

ZooKeeper keeps the records of fuzzing job and task information in consistence with their state. The broker inserts fuzzing jobs and tasks into the Apache HTTP server, and creates the corresponding znodes in ZooKeeper. In ZooKeeper, a parent znode is established to represent one fuzzing job, under which there are many child znodes created for the split fuzzing tasks. Both parent znodes and child znodes contain related information. Specifically, each parent znode contains information if the job is finished or not. Each child node has the following information: task is executed or not, task is downloaded or not, task execution commands.

The broker can get informed from ZooKeeper whether one job is finished or not. If the job is finished, the broker can get fuzzing results from its children. And ZooKeeper provides resources with information of unexecuted fuzzing task (if one task is unexecuted, the resource will download and execute it).

3.1.4 Resources

Generally, the network resources are invited by clicking links published by the broker, or other methods around. They will be given the resource-side code, which is an executable binary program. If resources join the fuzzing process, they will first be directed to connect ZooKeeper, where they can obtain unexecuted fuzzing tasks information. In the following steps, they will go directly to the Apache HTTP server and download corresponding unexecuted tasks indicated in ZooKeeper's fuzzing task znodes. Fuzzing will be completed in the local resources automatically right after fuzzing tasks are downloaded successfully. Also, fuzzing results are automatically sent back to the task znodes in ZooKeeper.

Due to the fact that there might be many remote resources to pick the same fuzzing task at the same time in the network, race conditions [60] are a big problem. One fuzzing task may be executed by many remote resources, while the others are left unattended. To prevent occurrence of this problem, each remote resource that wants to obtain a fuzzing task is required to establish a lock node as a child under the fuzzing task node. All remote resources requesting the fuzzing task are lined up and each remote resource obtains the lock in the order of request arrival time. By comparing sequence numbers, the remote resource that has the lowest sequence number will be allocated the fuzzing task. At the same time, other remote resources failing to obtain this fuzzing task will be directed to fetch other unexecuted fuzzing tasks in this job directory or other job directory.

3.2 System Functionalities

We mainly have three steps to achieve in this thesis: splitting one fuzzing job into many fuzzing tasks, inserting fuzzing jobs and tasks, executing fuzzing tasks and sending back fuzzing results. Splitting a fuzzing job is accomplished by developing many different fuzzing tasks on one application. To split one fuzzing job, we create many programs to fuzz the application with the help of SPIKE. Inserting fuzzing jobs and tasks is attended by the broker. Executing fuzzing tasks are handled by free remote resources in the network, and also these remote resources will send back fuzzing results to the ZooKeeper, which will be contacted by the broker to fetch job fuzzing results. Figure 3.2 presents the whole process.

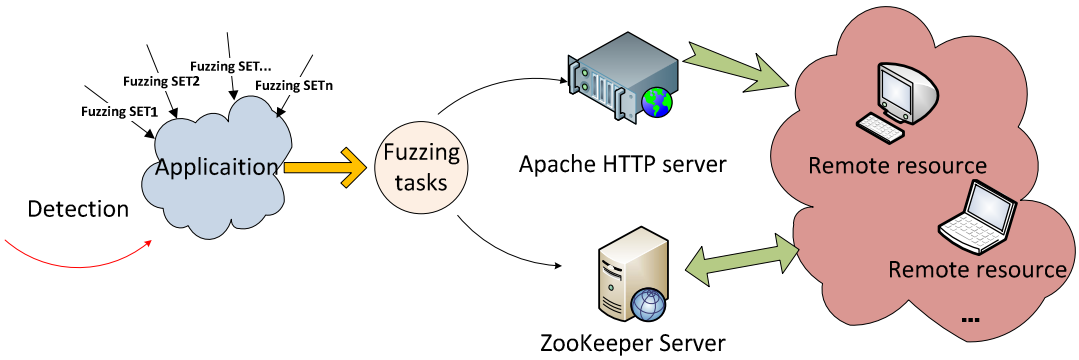


Figure 3.2 Process of fuzzing one fuzzing job

3.2.1 Split one fuzzing job

Following the methodology of SPIKE described in 2.2.2, the first thing to do is implementing SPIKE to create many fuzzing tasks when a fuzzing job is received from the client. We assumed that each task fuzzes a subset of the input fields. Basically, each fuzzing task is composed of two processes: one process is to enable feeding pre-constructed data to the input space automatically; the other process is to detect any abnormalities happened when the application is running.

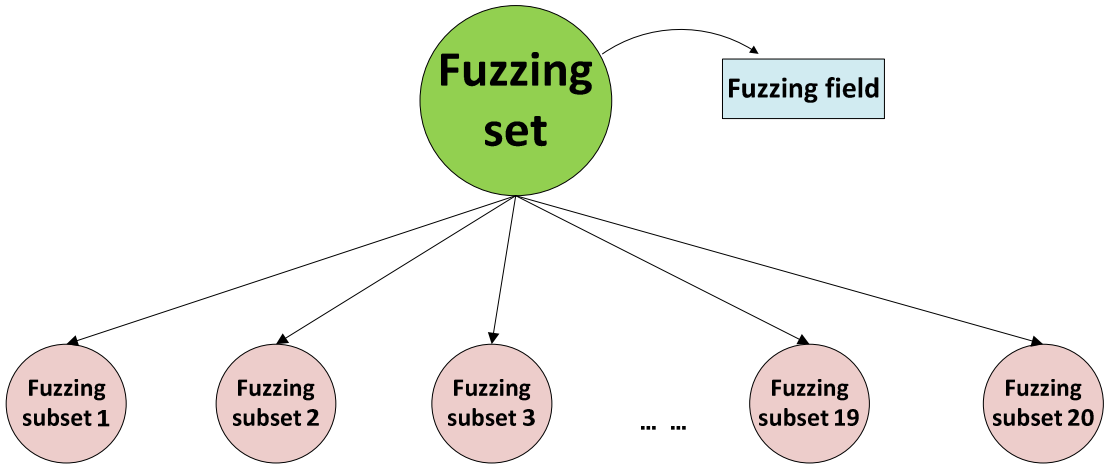
First, the input is analyzed to find out which part is fixed, and which part is variable. Those variable parts are the targets to fuzz. For example, one program called palette that requires a .png file as input. By studying the specification of .png file, we learned that each .png file starts with fixed 8-byte signature, thus it is not necessary to fuzz this field, because the program itself may simply regard files with different signatures as false and then reject them. On the contrary, IDAT [28] chunk in a .png file contains the image data that can be varied in order to generate different .png files. So fuzzing IDAT chunk is a good choice. If the fuzzing object is too large, we can simply create a

script (.spk file) to contain part of the fuzzing data.

To detect aberrant behavior, the application is triggered, and begins to accept data stream from the fuzzing process. Any abnormality in the execution process will be detected by status change. Eventually, executable binary files generated from detecting fuzzing application's abnormalities will be inserted in the HTTP server and recorded in ZooKeeper server, available for remote resources to download and execute.

Considering that there are many free network resources, we decide to allocate each fuzzing task within a certain time of work load which can be configured in relation to demand. In this grid fuzzing system, we just allocate 30 minutes for each fuzzing task. If the fuzzing task is out of time, its state will be reset as the initial state, so that other remote resource can pick this task to fuzz. Imposing a constant time on a fuzzing task provides convenience to quantitatively manage task state by the broker, guarantying fuzzing efficiency.

To achieve this, we need some strategies to divide one fuzzing job. First, we simply develop the fuzzing program to fuzz some fields of the input space. And then we divide the fuzzing set into fuzzing subsets, which are used in the same fuzzing program to fuzz the specified fields. By evaluating the time one fuzzing subset takes to finish fuzzing, we can adjust the number and volume of the fuzzing subsets as needed. Figure 3.3 gives a simple example how the set is split. The fuzzing set contains 1000 numbers, and it is split into 20 subsets, with 50 numbers in each subset. All fuzzing subsets work with the same fuzzing model as the fuzzing set.



Fuzzing set: {1,2,3,4,5,... ... 1000}
Fuzzing subset 1: {1,2,3,4,... ... 50}
Fuzzing subset 2: {51,52,53,... ... 100}
Fuzzing subset 3:{101,102,103,... ... 150}
... ...
Fuzzing subset 20: {951,952,953,... ... 1000}

Figure 3.3 Generate fuzzing subsets

3.2.2 Insert fuzzing jobs & tasks

After successfully generated fuzzing tasks (executable binary files), the next step is inserting executable files (fuzzing jobs and fuzzing tasks) into the Apache HTTP server and creating corresponding znodes in ZooKeeper.

The Apache HTTP server cooperates with ZooKeeper to manage fuzzing jobs and fuzzing tasks. First, the broker will always check how many fuzzing job directories in the local environment, and then create new fuzzing job directories in the Apache HTTP server if they are not fully covered.

Similar things will be done in ZooKeeper simultaneously, and new fuzzing job znodes will be created. Then fuzzing tasks under each fuzzing job will be sent to the corresponding fuzzing job directory in the Apache server. In ZooKeeper, these fuzzing tasks are deemed as children of one fuzzing job, thus under the parent znode of a fuzzing job, child znodes of fuzzing tasks are created.

Considering extensibility of the fuzzing system, the broker always checks each fuzzing job directory in the local environment to see if there are new fuzzing tasks added into the current directory. If so, the broker will pick up them and place them in the same way as described above, and Zookeeper will create znodes for them as well. This is quite an important attribute that it spares time and space for creating extra fuzzing

tasks, so that the broker can fuzz more objects or add new fuzzing combinations if new requirements are requested from clients, avoiding inadequacy in the process of fuzzing and making fuzzing process more flexible. From the practical point of view, this property guarantees extensibility of the gird fuzzing system.

3.2.3 Execute fuzzing tasks

The execution of fuzzing tasks is performed by the remote resources in network. Those free network resources that are willing to contribute fuzzing process will be allocated with unexecuted fuzzing tasks. To avoid repeated downloading, once the resource has downloaded one fuzzing task, ZooKeeper would update data information of fuzzing task by indicating that it has been downloaded, which in return helps other resources to decide which task is available.

As described in 3.1.4, concurrence happens when one task is picked up from HTTP server by one resource, but the task state in ZooKeeper is not updated promptly. As a result, more than two remote resources thought they obtained a different fuzzing task, but actually they are racing for the same task. As there are thousand millions of resources in the network, consequently, one fuzzing task may be downloaded and executed by many remote resources at the same time, while other fuzzing tasks are still pending for remote resources' attention. To prevent this kind of problem, a lock can be added to each fuzzing task node. Whenever one resource wants to pick up a locked fuzzing task which has previously been taken away by another remote resource, the fuzzing grid system will automatically direct this resource to deal with another new fuzzing task available.

After execution of fuzzing tasks is completed by remote resources, the fuzzing results will be feedback to ZooKeeper server. If one fuzzing task failed to be executed due to special situations, such as interruption, abnormal exit, the remote resource did not need to execute the fuzzing task again, because the fuzzing task state would be updated by the broker if it was out of time. Another remote resource in the network would pick the failed fuzzing task for execution. When the failed remote resource managed to work for fuzzing service again, it will be allocated with a new fuzzing task.

Chapter 4 Practical Implementation

Thus far, we have given an overview of the grid fuzzing system, and introduced its main functionalities. For deeply exploit the grid fuzzing system, in this chapter, we are going to discuss in detail how those components (ZooKeeper, Apache HTTP server, broker and remote resources in the network) of the grid fuzzing system work, and how they cooperate consistently to complete fuzzing jobs.

4.1 ZooKeeper

To use ZooKeeper server, we must get acquainted with how it is configured and how it starts to work.

After downloading and decompressing ZooKeeper, a configuration file zoo.cfg is created with the following lines:

```
tickTime=2000;
dataDir=/Desktop/ZooKeeper/zookeeper-log;
clientPort=2181
```

TickTime is the basic time unit in milliseconds used by ZooKeeper, which is used to do heartbeats and the minimum session timeout will be twice the tickTime. DataDir is the location to store the in-memory database snapshots and, unless specified otherwise, the transaction log of updates to the database. ClientPort is the port to listen for client connections [12].

And then, ZooKeeper is started by calling the shell script in the terminal:

```
bin/zkServer.sh start
```

Figure 4.1 shows the status of successfully started Zookeeper server.

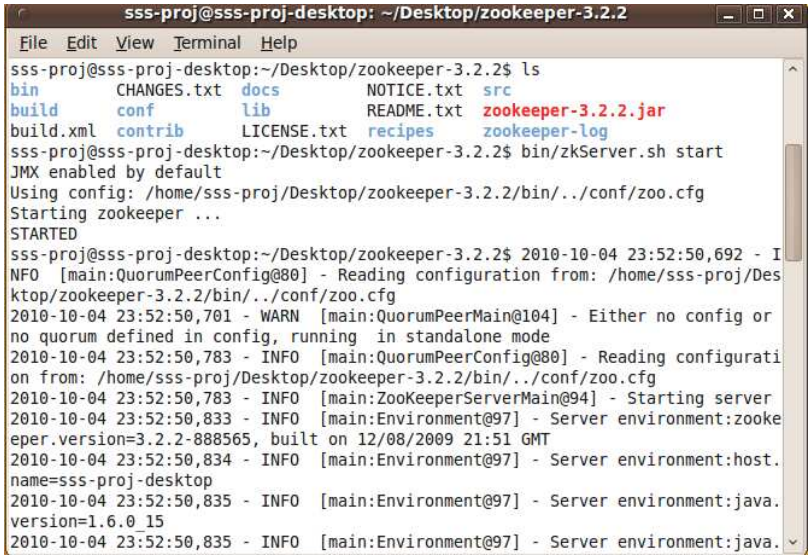


Figure 4.1 Start ZooKeeper server

To use the ZooKeeper service, an application must firstly be instantiated as an object of ZooKeeper class. All the interactions are done by calling specific methods of this class. In this project, the broker and remote resources in network will communicate with ZooKeeper directly, so that they contain object of the ZooKeeper class. If the broker or resources in the network would like to connect the ZooKeeper, the following command is executed:

bin/zkCli.sh -server 127.0.0.1:2181

Figure 4.2 demonstrates a connection of the broker/resources to the ZooKeeper server.

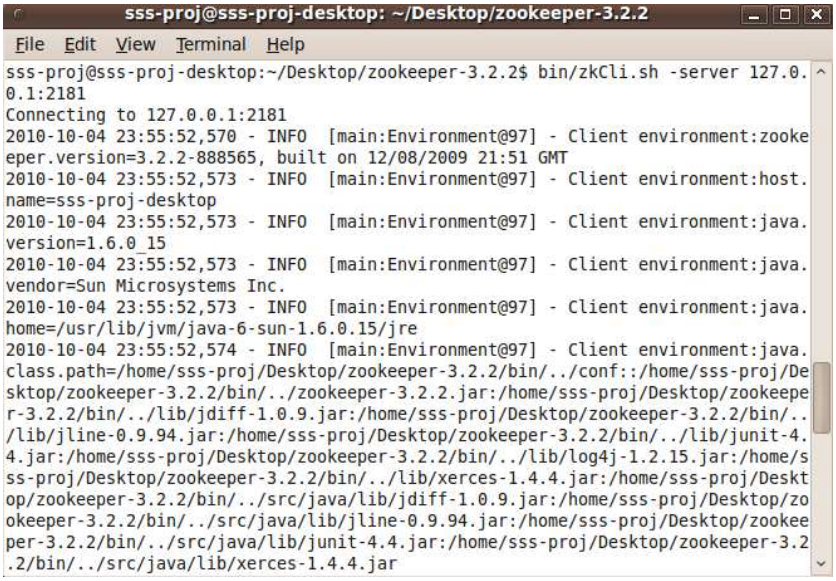


Figure 4.2 Connect to ZooKeeper

In the grid fuzzing system, the broker connects ZooKeeper to insert fuzzing jobs and tasks, or get fuzzing results. And remote resources connect ZooKeeper to obtain fuzzing tasks and send feedback to it. To make the connection go on automatically, the broker and remote resources are designed to connect ZooKeeper using the following Java code:

```
final static String ZookeeperServer = "127.0.0.1:2181";
zkoperator.connect(ZookeeperServer);
```

As mentioned in 2.3, the development of this system is based on a local environment, so that the ZooKeeper is connected by sending connection request to 127.0.0.1, saving the effort to send and receive requests. In a distributed system, ZooKeeper server's address is the server's IP address.

4.2 Broker

After fuzzing tasks are ready, the primary work of the broker is to insert them into the Apache server, while creating the corresponding znodes in ZooKeeper. Specifically, the broker has the following functionalities:

- Create the fuzzing job directory in Apache server if a fuzzing job has not been established before; the following code gives an instance how to create a fuzzing job directory:

```
File dstdir=new File(Dst_JobDir);
success = dstdir.mkdir();
```

And Figure 4.3 shows the result, in which a directory ("palettefuzz") is successfully created.

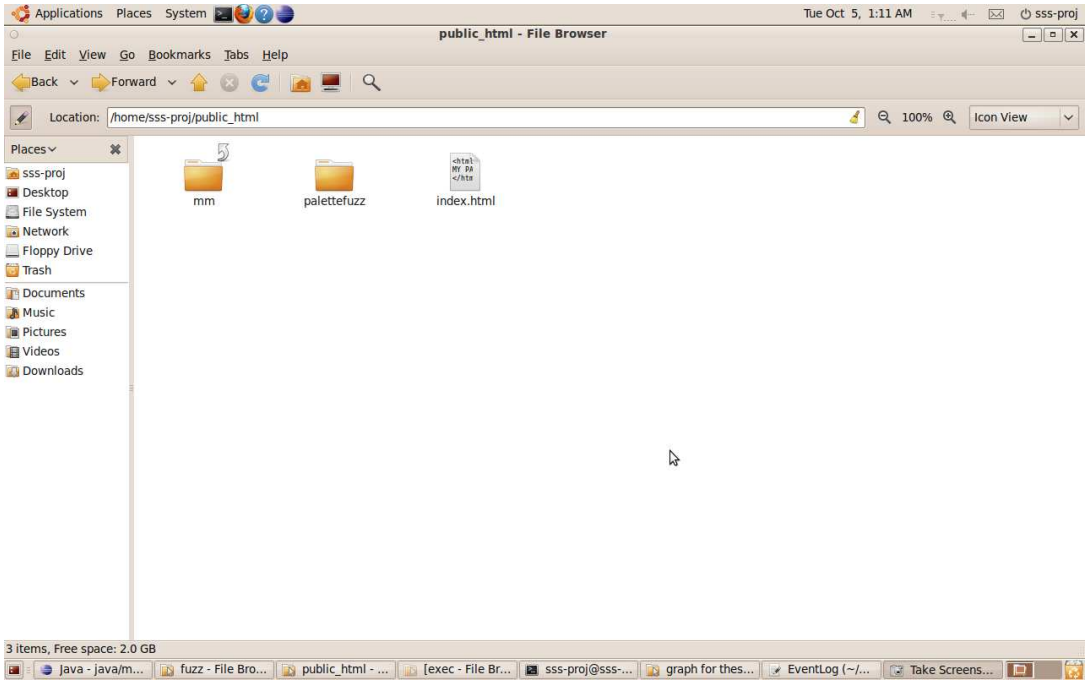


Figure 4.3 Create job directory in Apache HTTP server

- Upload fuzzing tasks of one fuzzing job into Apache server; the following code is used to achieve this process and figure 4.4 shows that two fuzzing tasks ("gopherd" and "palfuzz_task1") are uploaded successfully in the "palettefuzz" job directory.

```
public static void taskInsert(File localFileName, File desFileName) throws IOException
{
        InputStream         inStream = new FileInputStream(localFileName);;
        OutputStream        outStream = new FileOutputStream(desFileName);
        byte[] buf = new byte[1024];
        int len;

       while ((len = inStream.read(buf)) > 0)
        outStream.write(buf, 0, len);

       inStream.close();
       outStream.close();
 }
for (int j=0;j<locdir.listFiles().length;j++)
{
```

```
    TaskNode = locfolder.listFiles()[i].getName()+"/"+fuzztask.taskname;
    File localTask=new File(localFuzzdir+TaskNode);
    File dstTask=new File(dstFuzzdir+TaskNode);
    jtinsert.taskInsert(localTask, dstTask);
  }
```

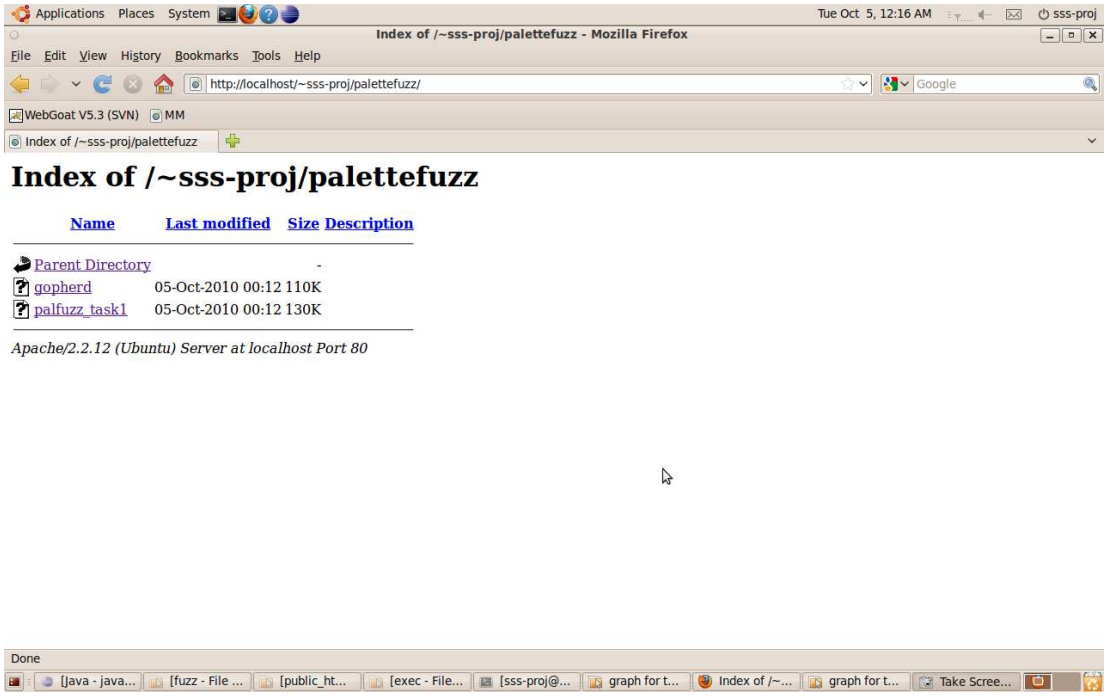


Figure 4.4 Insert job and tasks into Apache HTTP server

- If new fuzzing tasks are added, they can also be inserted into the Apache server, and be recorded in ZooKeeper; this is performed by the following code and a successful instance is shown in figure 4.5, where a new task ("halflife") is inserted into the "palette" job directory.

```
for(int k=0; k<TaskNum; k++)
{
  File tmpDstFile=new File(Path_Fuzzingtask[k]));

  if(!tmpDstFile.exists())
  {
    TaskNode = TaskPath;
    File localTask=new File(localFuzzdir + TaskNode);
    jtinsert.taskInsert(localTask, tmpDstFile);
  }
}
```

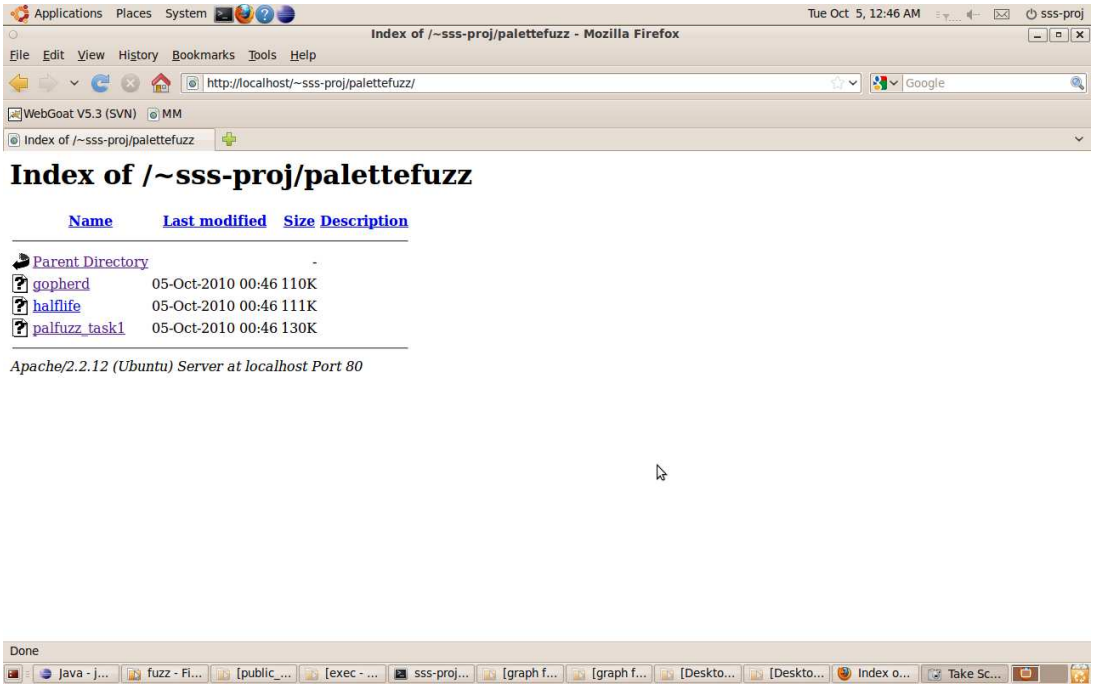


Figure 4.5 Insert new tasks

- If new fuzzing jobs are added, new directories will be established in the HTTP server and the corresponding tasks will be inserted into the Apache HTTP server, and recorded in ZooKeeper. Figure 4.6 shows the results.

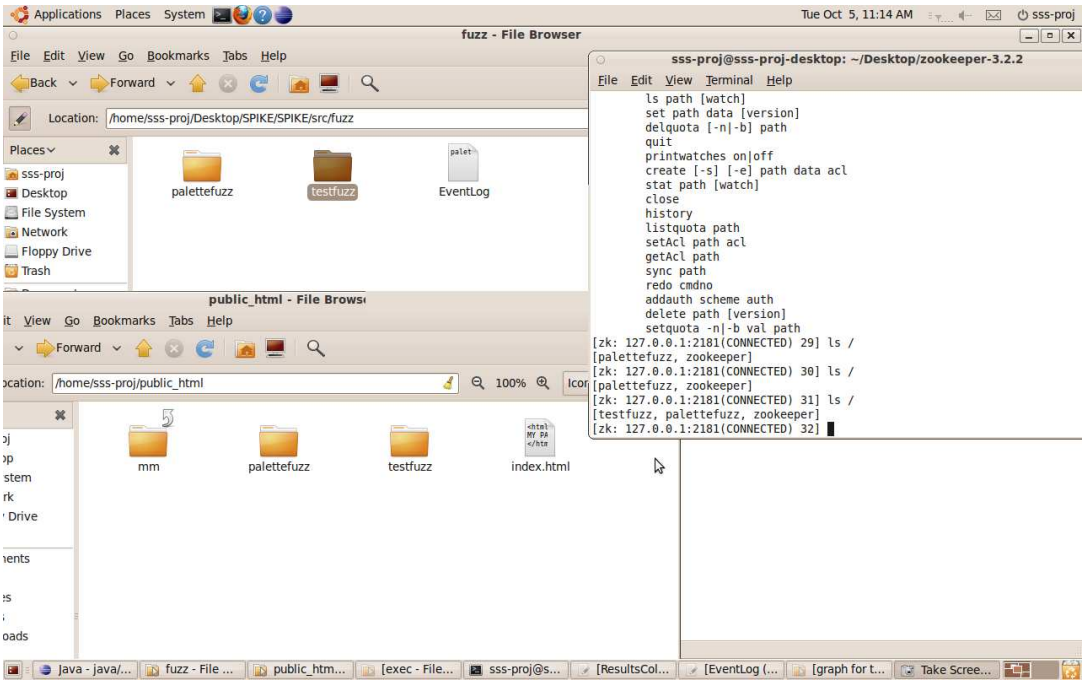


Figure 4.6 Instance of adding one new fuzzing job

- Connect the ZooKeeper server, and record fuzzing job and fuzzing tasks information. Figure 4.7 shows the fuzzing task state in ZooKeeper after being inserted.

```
String info2=TaskInformation;
byte[] taskinfo=new byte[info2.length()];
taskinfo=info2.getBytes();

zkoperator.create(TaskPath,taskinfo);
```

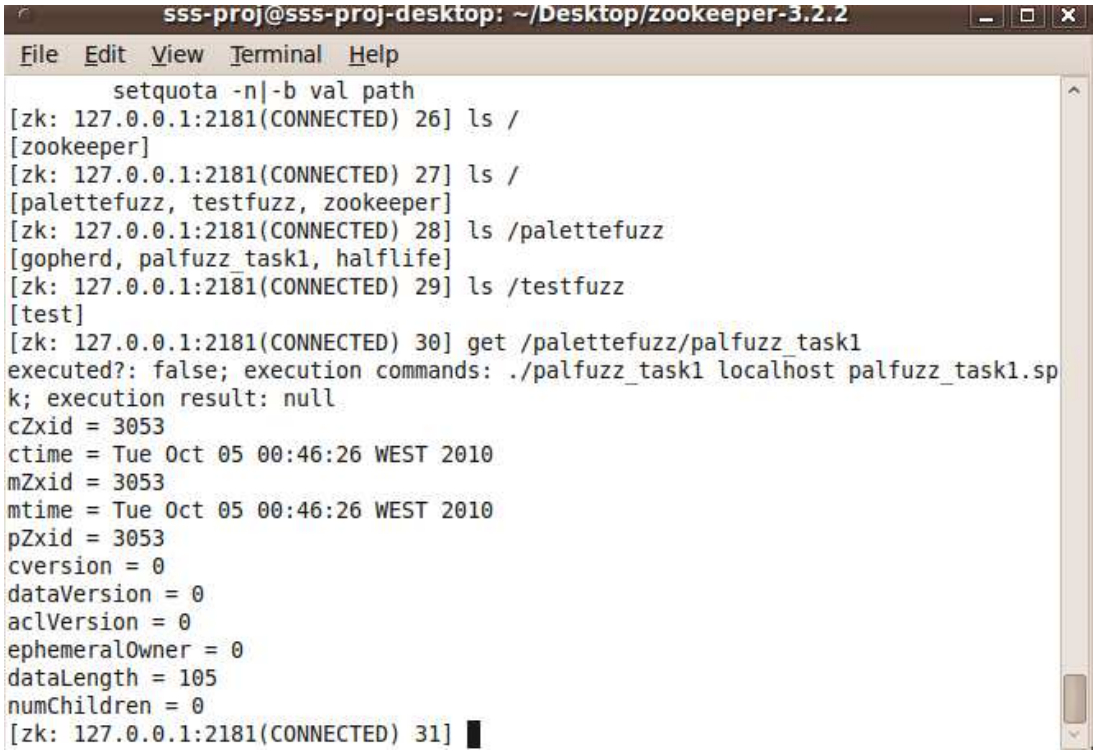


Figure 4.7 Record job and task in ZooKeeper

- Monitor those fuzzing tasks under each fuzzing job. If one fuzzing task is out of time, the task will be reset in the ZooKeeper, and other remote resource can compete for this fuzzing task again. Meanwhile, the monitor checks if all fuzzing tasks results are available. If so, fuzzing job node should be updated by indicating that the fuzzing job is finished. Figure 4.8 illustrates the node state.

```
if (TaskDataInfo.contains(DownLoadInfo))
        if(Day == 0)
        {
         if(Hour == 0)
         {
                if (Min >= 30)
                        ResetTask = 1;
        }else ResetTask = 1;
     }else ResetTask = 1;

if(Dataversion >= 2)
if (!TaskDataInfo.contains(ExePrefix))
                if (!TaskDataInfo.contains(DownLoadInfo))
                        taskcount++;
                        }
        if(taskcount == AllTasks.size() && taskcount != 0)
```

zkoperator.setdata(JobPath, JobFinished);

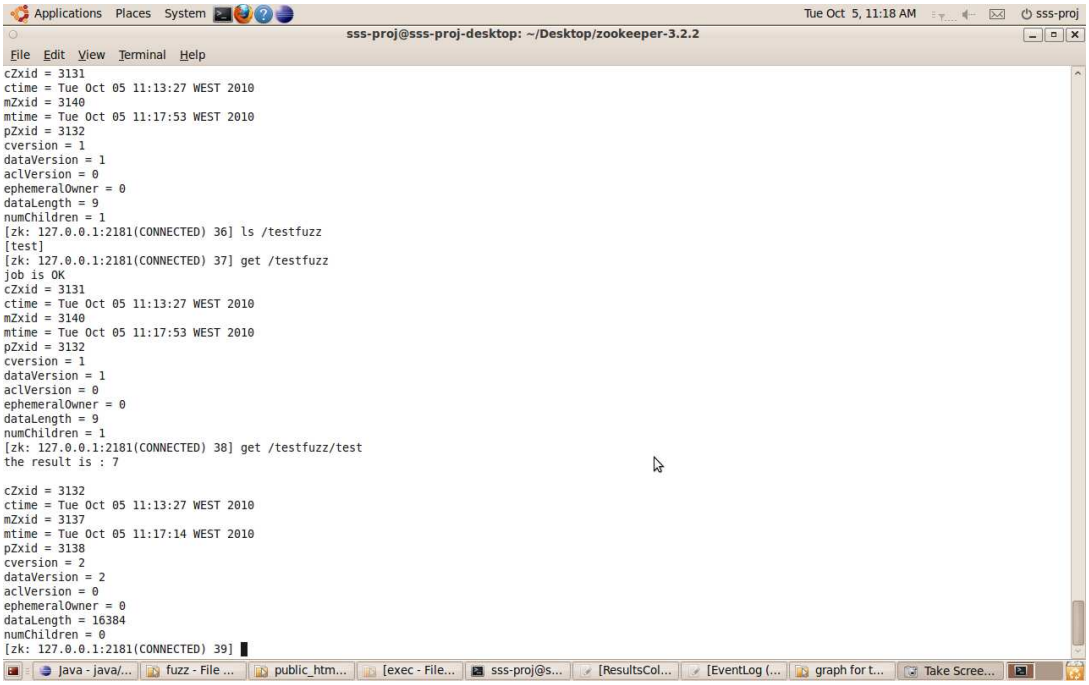


Figure 4.8 Upload fuzzing job node

- Check which fuzzing job is finished by connecting ZooKeeper server, and fetch results recorded in fuzzing tasks nodes of ZooKeeper server. Execution results will be kept in a ResultsCollec.txt file, which will be sent to corresponding clients. Figure 4.9 shows the process.

```
if(JobDataVersion == 1)
 {
    TaskData = zkoperator.getData(Taskpath);
    TaskResult.write(tasks.get(tasknode)+ " result "+": ");
   TaskResult.write(new String(TaskData)+"\n");
}
```

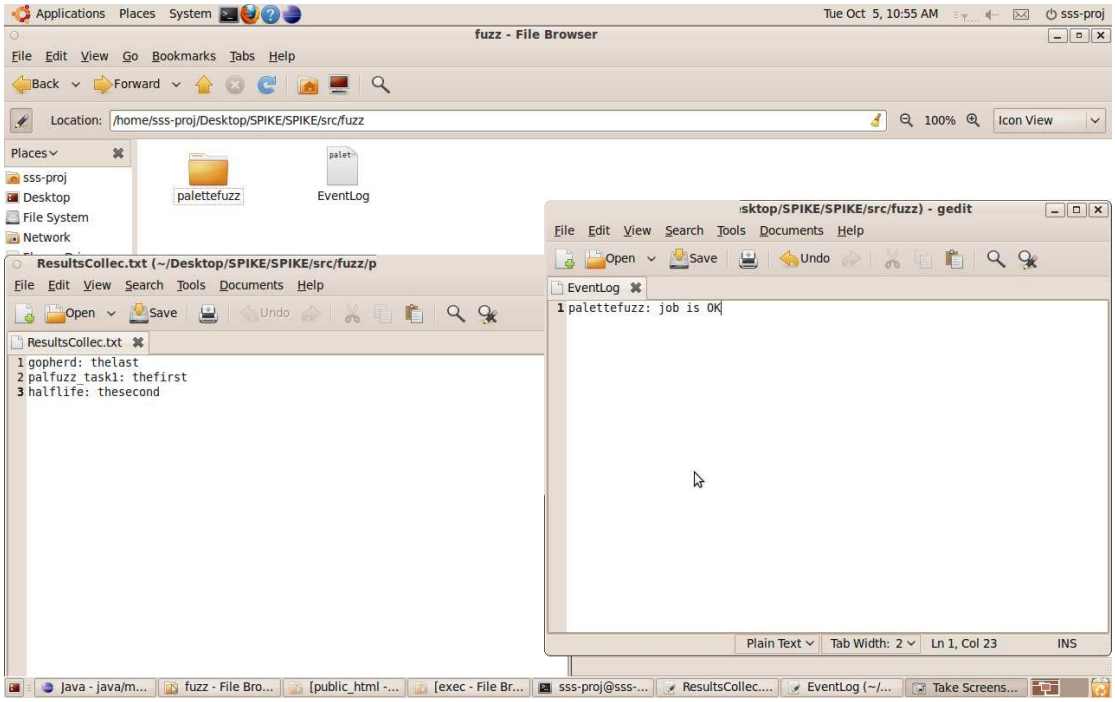


Figure 4.9 Write fuzzing task results to file

- After fetching all results of one fuzzing job, the directory will be deleted in HTTP and ZooKeeper server, sparing space for other fuzzing jobs. Figure 4.10 shows the result after deleting a job.

```
tasks = zkoperator.getChild(Jobpath);
for(int tasknode = 0; tasknode < tasks.size(); tasknode++)
 {
      zkoperator.Delete(Taskpath);
      fuzztask.delete();
  }

fuzzjob.delete();
zkoperator.Delete(Jobpath);
```

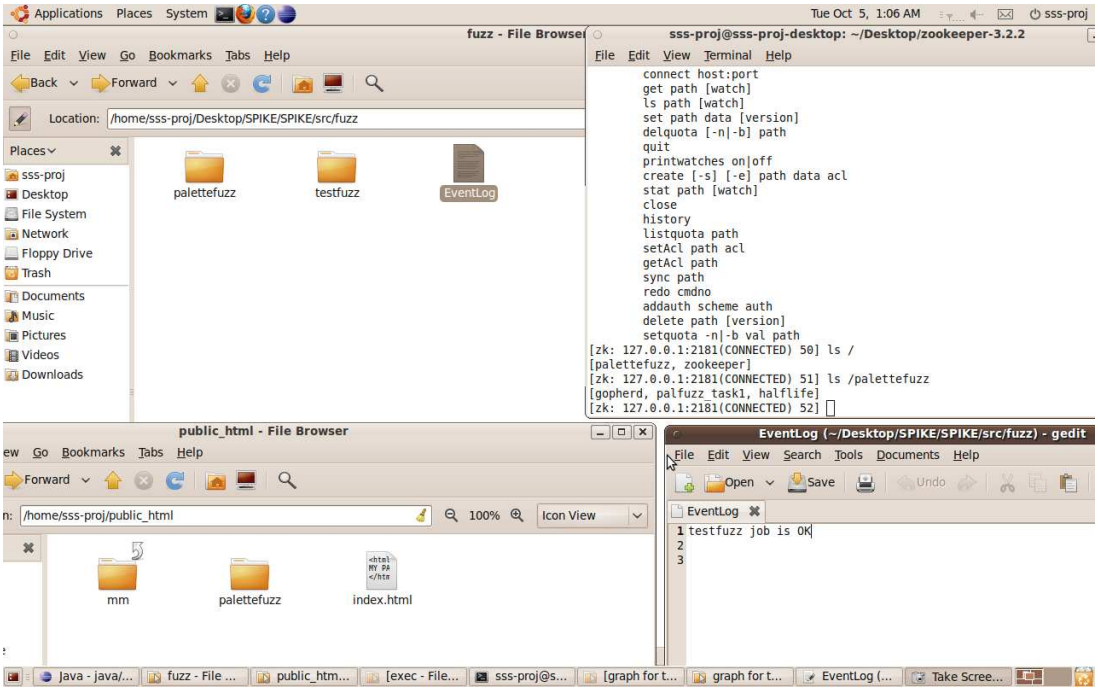


Figure 4.10 Delete job and tasks in HTTP and ZooKeeper servers

4.3 Resource

The resource's work is to obtain unexecuted tasks by downloading them from the Apache server. The functionalities are listed below:

- Connect the ZooKeeper, get the unexecuted fuzzing job;

```
ch=zkoperator.getChild("/");
        for(int i=0;i<ch.size();i++)
    {
            data=zkoperator.getData("/"+ch.get(i));
            datainfor=new String(data);
            if(datainfor.contains("false"))
             {
                 fuzzjob=ch.get(i);
                 break;
             }
    }
```

- Pick one unexecuted task from a child znode of one fuzzing job;

```
ch=zkoperator.getChild(Job);
        for(int i=0;i<ch.size();i++)
```

```
{
        data=zkoperator.getData(Fuzzingtask[i]);
        datainfor=new String(data);
        if(datainfor.contains("false"))
        {
            fuzztask=ch.get(i);
            break;
         }
   }
```

- Check if there are other remote resources trying to pick up the same fuzzing task;

```
zkoperator.CreateLock(LockNode);
LockRequestQueue = zkoperator.getChild(LockNode);

for(int request = 0; request < LockRequestQueue.size();request++)
{
    if(Sequence > NodeSeq[request])
        Seqtmp =NodeSequence[request];
    if(Sequence!= Seqtmp) break;
 }
if(Seqtmp == Sequence)
TaskAvailable=1;
```

- Download unexecuted task from Apache server, and update information in the ZooKeeper; Figure 4.11 shows the new state after a fuzzing task is downloaded.

```
public static void downloadUrl(String fAddress, String localFileName, String destinationDir)
    {
        InputStream         inStream = null;
        OutputStream        outStream = null;
        URLConnection       uCon = null;

        URL Url=new URL(fAddress);
        byte[] buf= new byte[size];
        int ByteRead,ByteWritten=0;

        outStream = new BufferedOutputStream(new
                    FileOutputStream(destinationDir+localFileName));

        uCon = Url.openConnection();
        inStream = uCon.getInputStream();
```

```
        while ((ByteRead = inStream.read(buf)) != -1) {
            outStream.write(buf, 0, ByteRead);
            ByteWritten += ByteRead;
            }

        System.out.println("Downloaded Successfully.");
        System.out.println("File name:\""+localFileName+ "\"\nNo ofbytes :" + ByteWritten);
    }
```

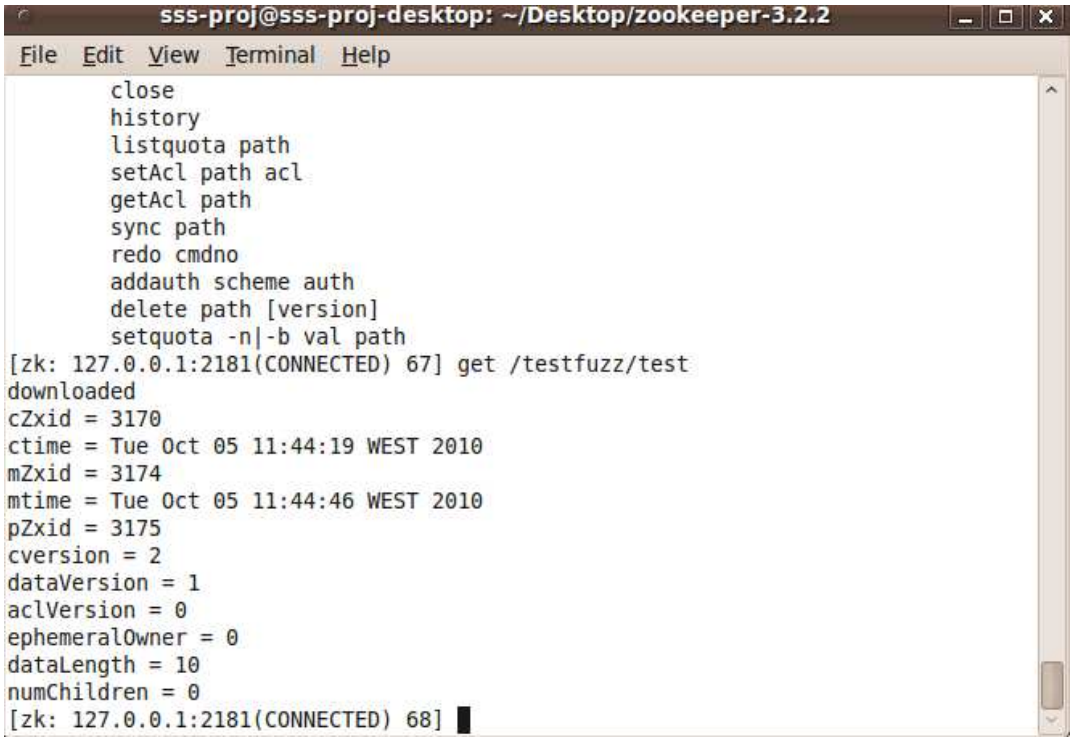


Figure 4.11 Download a fuzzing task

- Execute the task and obtain the execution results;

```
public byte[] execCmd(String path) throws IOException, InterruptedException
{
        String cmdExec=path;
        ProcessBuilder pb = new ProcessBuilder("bash", "-c" ,cmdExec);
        pb.redirectErrorStream(true);
        Process shell = pb.start();
        InputStream shellIn = shell.getInputStream();
        int shellExitStatus = shell.waitFor();
        System.out.println("The Exit Status is:" +shellExitStatus);
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();

        // close the stream
```

```
            try {shellIn.close();} catch (IOException ignoreMe) {}*/
            int nRead;
            byte[] data = new byte[16384];

            while ((nRead = shellIn.read(data, 0, data.length)) != -1)
              buffer.write(data, 0, nRead);

            buffer.flush();
            return data;
        }
```

- Upload the execution results into the ZooKeeper; Figure 4.12 illustrates it.

```
zkoperator.setdata(TaskNode, ResourceServ.execCmd("test"));
```

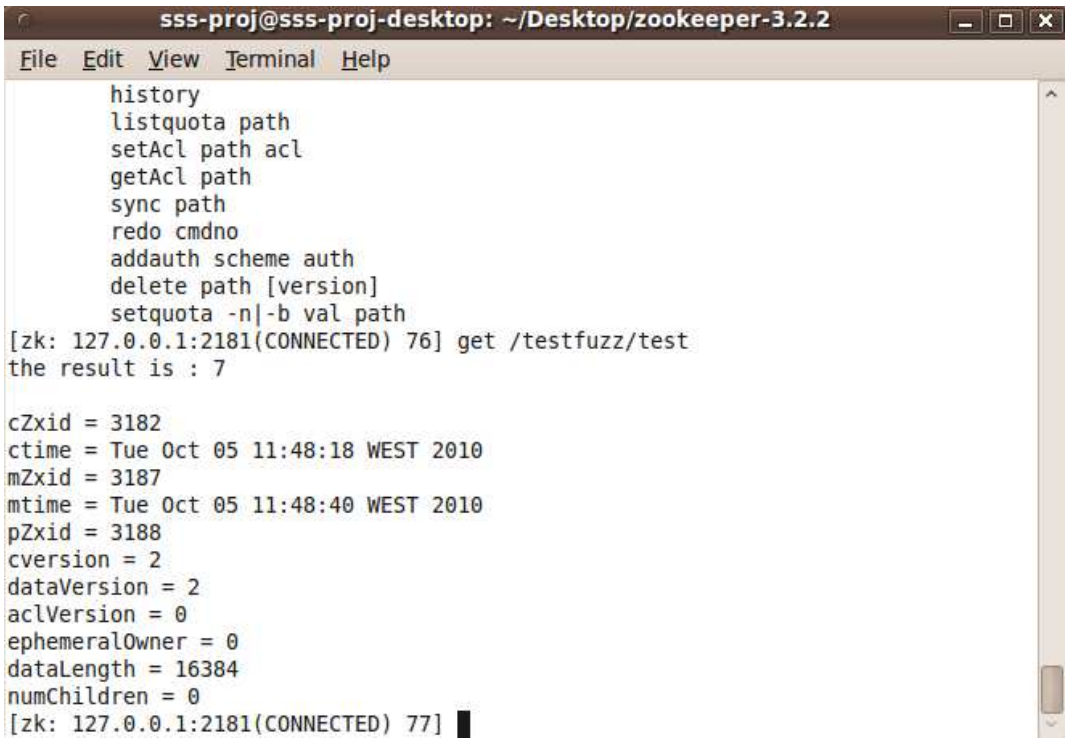


Figure 4.12 Upload fuzzing result

Chapter 5 Testing and Results Analysis

After the system development was concluded, it was tested in a distributed environment. To show that our system can be used to test different applications, we fuzzed the following three applications:

- One program called palette that takes a .png file as input, and outputs a color-decreased .png file;
- A second other program called std, which takes three input parameters and then prints them on the screen.
- Another program we fuzzed was json. It takes defaulted-format input, and outputs the extracted volumes from the keys.

5.1 Palette fuzzing

For executing palette, we have to provide it with one .png [28] file. Following the SPIKE methodology, the first step is to understand the format of a .png file. The .png file has mainly four chunks: IHDR, PLTE, IDAT, and IEND. Theoretically, all data fields can be fuzzed. Yet some fields are not necessary to be fuzzed, because of the fact that the program itself has the ability to reject obviously false files (e.g., those that do not contain an IHDR chunk). Another reason is that fuzzing some chunk requires more data. For example, fuzzing an IDAT chunk needs random length and random value, which means that after thousands of attempts we might find some vulnerability, but it can take a very long number of attempts. To fuzz palette more efficiently, we can choose those fields that have a fixed and low number of bytes, but identifying important properties of the .png file, like width, height, bit depth, and color in IHDR chunk.

We created two pictures—pic.png and simple.png. The first step is to create many fuzzing tasks. We use SPIKE library, SPIKE head files and some related SPIKE C files to help develop each fuzzing task. For experimentation, we just fuzzed four fields of the input. Each task fuzzed different fields. For example, picpalette1 fuzzed the first two bytes of width in IHDR chunk; picpalette2 fuzzed the last two bytes of width in IHDR chunk…It is the same way around for other fuzzing tasks. After all fuzzing tasks are generated, we created a job directory named “palette” in Apache HTTP server, and then copied all fuzzing tasks (executable binary files) into the “palette” directory. Then, we connected to ZooKeeper, creating parent node “palette”, under which, all fuzzing tasks nodes are created. The “palette” node contains job state information. A fuzzing task node, like picpalette1, includes task state information and execution command. What needs to be noticed is the “palette” job directory includes picpalette fuzzing tasks and simplepalette fuzzing tasks.

Table 5.1 and Table 5.2 show the fuzzing results.

Fuzzing tasks	Time for fuzzing(min.sec)	Vulnerabilities found
picpalette1	2.44	0
picpalette2	2.38	0
picpalette3	3.06	2
picpalette4	10.39	0
picpalette5	7.55	0
picpalette6	8.37	2
picpalette7	9.18	0
picpalette8	8.10	0
picpalette9	7.58	1
picpalette10	8.40	0
picpalette11	9.04	2
picpalette12	10	0
picpalette13	10.16	0
picpalette14	8.35	0
picpalette15	9.21	0
picpalette16	9.24	0
picpalette17	10.57	2
picpalette18	10.41	2
picpalette19	9.40	0
picpalette21	9.17	1
picpalette23	7.29	1
average time	8.21	

Table 5.1 Picpalette fuzzing results

In the picpalette testing, some fields are repeatedly fuzzed. For example, picpalette21 fuzzed the last byte of width and height, and picpalette22 fuzzed the last byte of width and the penultimate byte of height. There are some overlaps between the two fuzzing tasks, and the vulnerability by chance happened in the overlapped fields. That is why both fuzzing tasks found the same vulnerability. After comparing these vulnerabilities, there are only 3 different ones found. In practical application, efforts must be taken to split a fuzzing job, preventing occurrence of overlapped fuzzing sets between fuzzing tasks. As a result, less time is consumed for broker to get fuzzing results. In the following simplepalette fuzzing test result table, those repeated results are eliminated.

Fuzzing tasks	Time for fuzzing(min.sec)	Fuzzed fields	Vulnerabilities found	Average time (Tav(min.sec))
simplepalette1	7.22	4	0	(1) 15.20
simplepalette2	7.37	4	0	
simplepalette3	5.02	4	0	
simplepalette4	4.27	4	0	
simplepalette5	4.13	4	0	
simplepalette15	3.17	4	2	
simplepalette7	0.49	3	0	(2) 0.45
simplepalette8	0.44	3	0	
simplepalette9	0.44	3	0	
simplepalette10	0.44	3	0	
simplepalette11	0.45	3	0	
simplepalette12	0.46	3	0	
simplepalette13	0.03	2	0	(3) 0.03
simplepalette14	0.03	2	0	
simplepalette16	0.03	2	0	
simplepalette17	0.03	2	0	
simplepalette18	0.02	2	0	
simplepalette19	0.03	2	0	

Table 5.2 Simplepalette fuzzing

We tested different numbers of fields. From the average fuzzing time, Tav(3) is 15 times of Tav(2), and Tav(4) is more than 7 times of Tav(2)(Table 5.2). We can conclude that if all tasks are tested in the same environment, time distribution of testing different fields would be linear. Based on this assumption, time spent for fuzzing different fields can be inferred, so that we can arrange fuzzing time corresponding to the condition of the network. In practical application, estimating execution time of a fuzzing task is very important, so that the broker can allocate an upper bound time for executing a fuzzing task, which in return helps to monitor the fuzzing task state.

The following snapshots figure 5.1, figure 5.2 and figure 5.3 represents the vulnerable inputs found in pic.png and simple.png files, which cause the program “palette” crash.

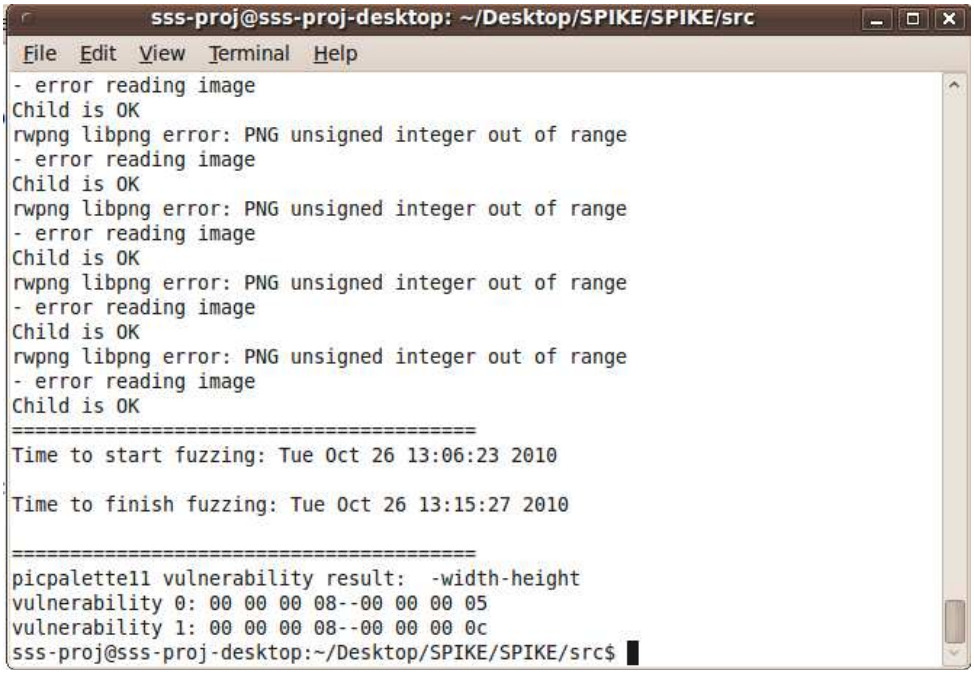


Figure 5.1 Picpalette11

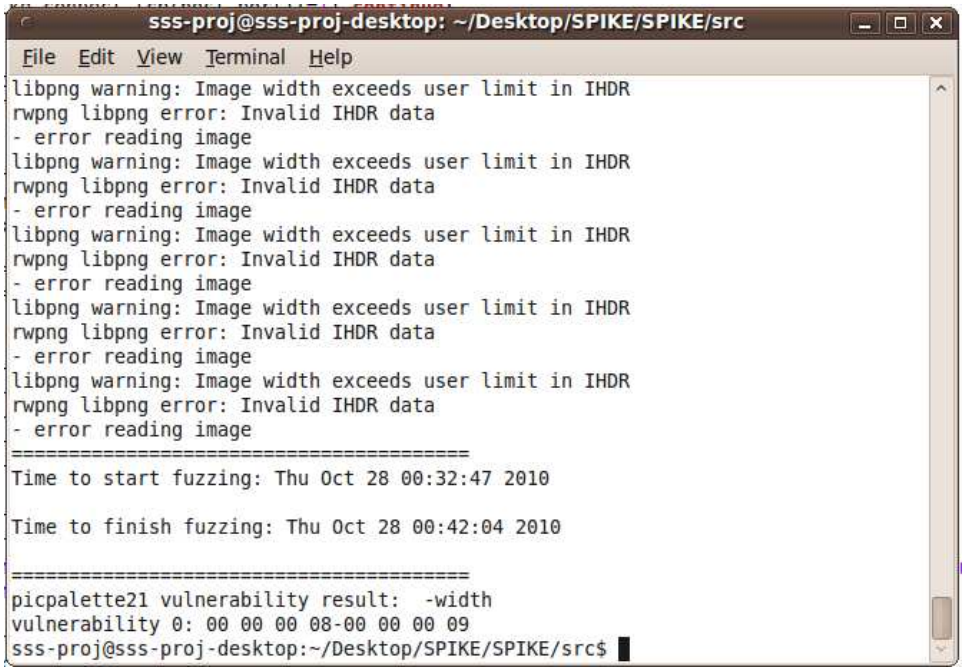


Figure 5.2 Picpalete21

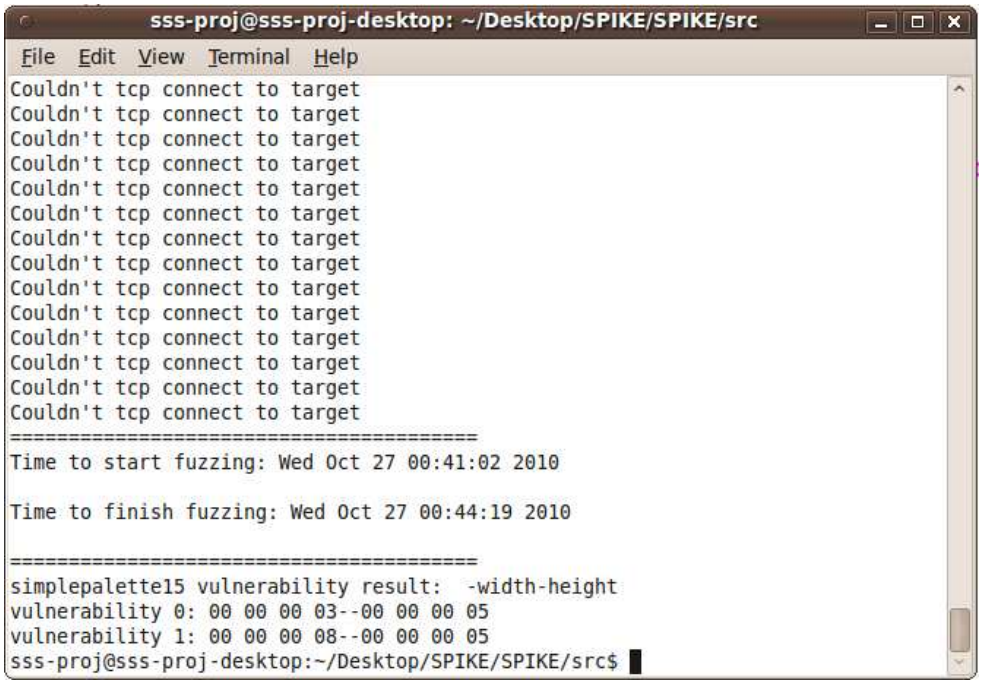


Figure 5.3 Simplepalette15

In figure 5.4, the fields of bit depth and color type were fuzzed, and then one vulnerable input was found. In this fuzzing task, fuzzing is designed to end when if one vulnerable input was found. Later, fuzzing task was modified in order to find if there are new vulnerabilities. But no vulnerability could be found any more. Nevertheless, the “01 00” is still deemed as vulnerability, because exception might happen at any time in this program.

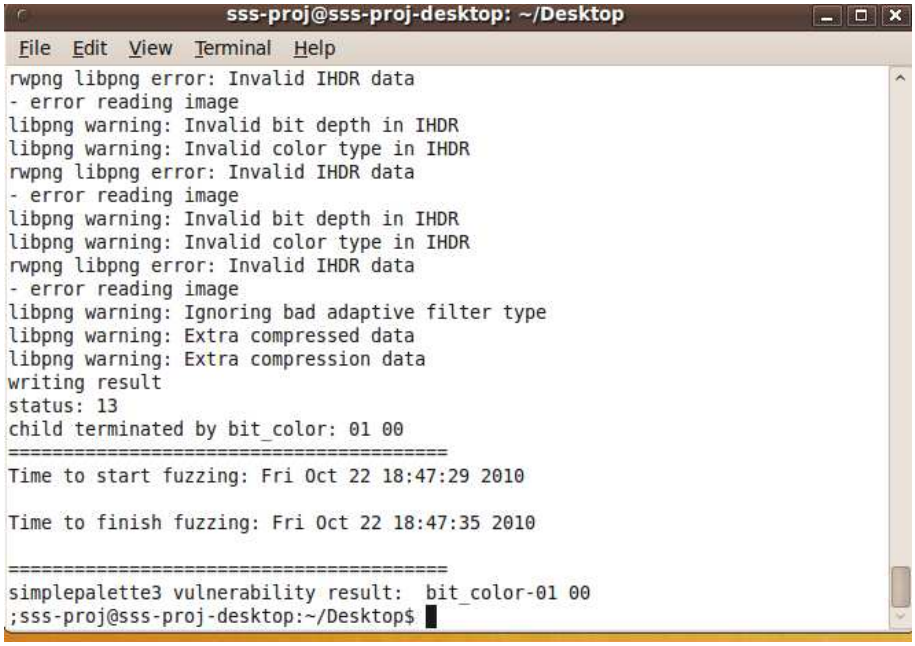


Figure 5.4 Simplepalette3

5.2 Std fuzzing

The other program we fuzzed is std. Std is a C program developed by us, which takes three strings as input, and the strings are printed on the screen. As long as the program only requires 3 parameters, we created a vector with a volume of 89 possible inputs from the keyboard. We picked fields 2, 3, 4, 6 to see if there is a crash caused by potential vulnerable inputs. All these fields will get a value from the vector. After the fuzzing programs are ready, we established a std directory in the Apache HTTP server, and then send all fuzzing programs into the std directory. Meanwhile, parent znode “std” is established in ZooKeeper, and the fuzzing task znodes are created under “std”. Fuzzing all std tasks is accomplished in several computers.

Table 5.3 shows the fuzzing result:

Fuzzing tasks	Test fields	Time (d.h.m.s)	Vulnerability
stdfuzz0	2	0.0.0.26	0
stdfuzz1	3	0.1.16.57	0
stdfuzz2	4	4.1.47.52	0
stdfuzz4	6	1.10.59.03	0

Table 5.3 Fuzzing std

The table shows that fuzzing 4 fields took more time than fuzzing 6 fields. This happens when the resource which is fuzzing the task with 4 fields has other processes going on in the system, hardware resource are not enough to support the

fuzzing task. In practical implementation, it is better to ensure that the network resources are free.

The table shows that although std was fuzzed during a time much larger than palette, several days in total, no vulnerabilities were found

5.3 Json fuzzing

Another program we fuzzed is called json. We borrowed the idea from Json [26, 27] data-interchange format. We developed the json program using C. The input space of the program requires a string with format “[Name: ***, Age: ***]”; “Name” and “Age” are the two keys. The fields with “***” are the volumes which could be combinations of characters. Volumes from the two keys are extracted and then printed in the console. But there is no specific restrictions on the input format in the program, thus it invisibly bury some vulnerabilities in the program. Like what we had done with “palette” and “std”, we created fuzzing tasks to fuzz json.c. Specifically, we mainly fuzzed two fields in the input space: volumes of name and age. For testing the name, 30 characters are traversed, and for testing the age, 14 characters are tried. Table 5.4 is collection of the fuzzing results.

Fuzzing tasks	Test fields	Time (d.h.m.s)	Vulnerability
jsonfuzz	2	0.0.0.5	1
jsonfuzz1	3	0.0.2.48	87
jsonfuzz3	2	0.0.0.2	0
jsonfuzz4	3	0.0.0.30	0
jsonfuzz5	2	0.0.0.9	0
jsonfuzz6	2	0.0.0.2	169

Table 5.4 Fuzzing json

Figure 5.5-5.7 shows the tasks that tested out some vulnerable inputs.

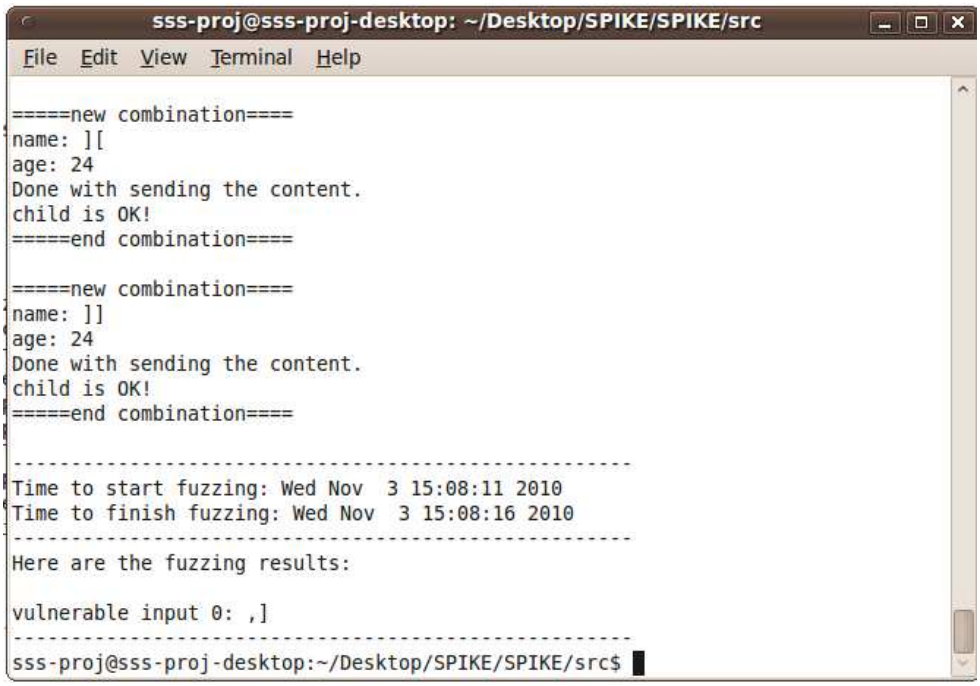


Figure 5.5 Jsonfuzz

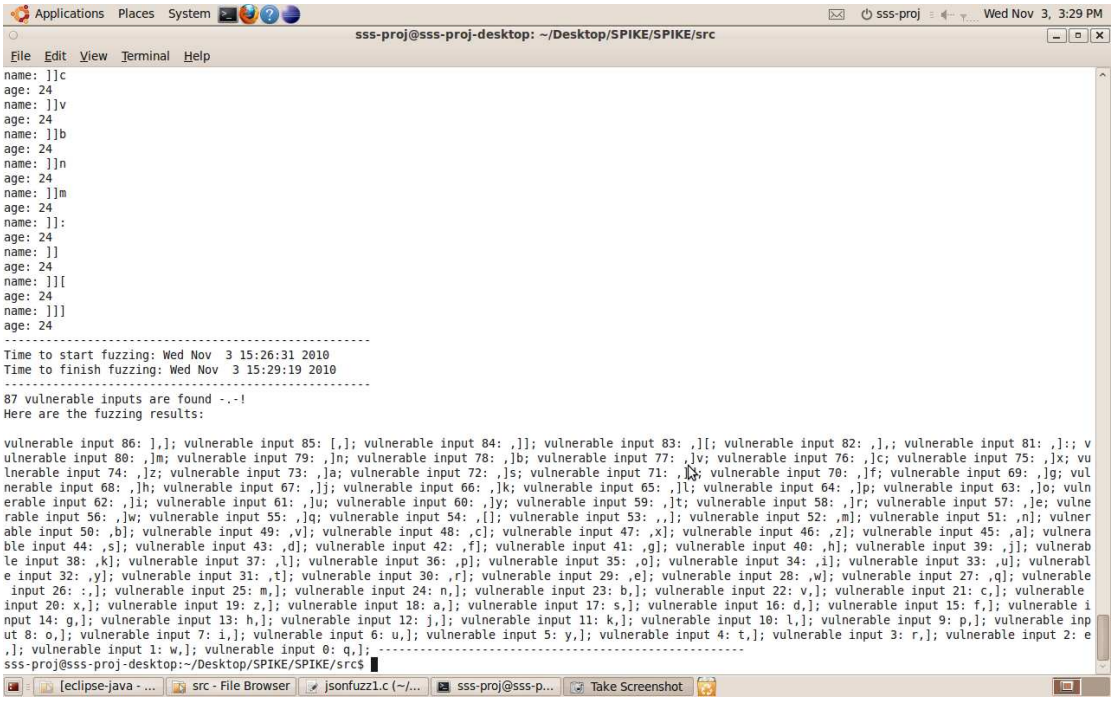


Figure 5.6 Jsonfuzz1

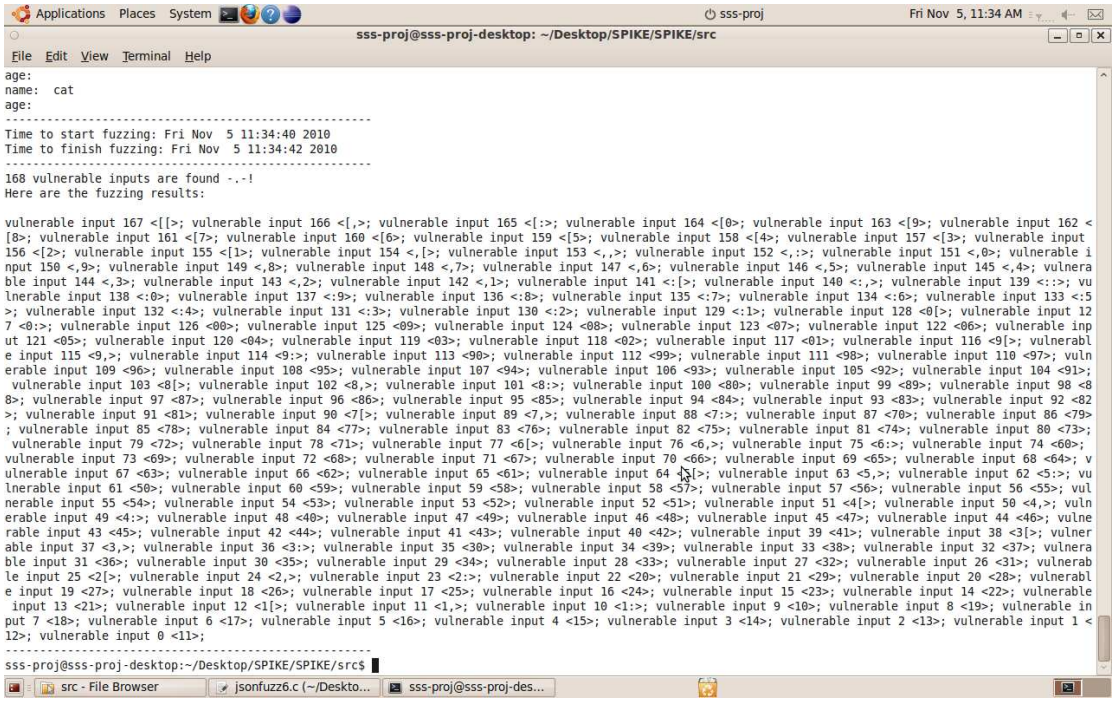


Figure 5.7 Jsonfuzz6

From the fuzzing results of json, we got clues to deal with vulnerabilities from the input information. The json1 has only one vulnerable input. By increasing the testing fields, we could find the vulnerable inputs in a generic way: the program is crashed by the three characters: “[”, “]” and “,”. To address these problems, these special characters must be carefully handled to prevent vulnerabilities in the program.

Chapter 6 Conclusion and Future Work

6.1 Summary

To promote testing applications efficiently, we proposed a grid computing environment for fuzzing, which splits testing one application into testing many fields of the application, and made use of network remote resources to test applications simultaneously. Generally, the grid fuzzing system has the following attributes:

- Divide one fuzzing job into many fuzzing tasks;
- New fuzzing jobs can be added into the system, and new tasks which are intended for existed fuzzing job can be inserted as well.
- ZooKeeper and Apache HTTP server cooperate to coordinate fuzzing service;
- Free network resources can download fuzzing tasks simultaneously from Apache HTTP server, execute fuzzing tasks respectively and feedback fuzzing results to ZooKeeper automatically;
- In order to prevent multi-remote-resources from executing the same fuzzing task, lock is designed in the fuzzing system to eliminate concurrence.
- Considering that fuzzing task can fail in remote resources due to link problem, download failure, etc., fuzzing task's state will be reset if fuzzing task state is not uploaded in a certain time after the task is allocated to one particular remote resource, so that other resources can fuzz this task again.
- Fuzzing job results in ZooKeeper are collected by the broker;
- For sparing more space for other fuzzing jobs, fuzzed jobs will be deleted by broker right after the job's fuzzing results are obtained.
- Each fuzzing job will be given a .log file for receiving fuzzing results from ZooKeeper;

By dividing one fuzzing job into many fuzzing tasks, we can arrange more fuzzing sets to test applications in a complete way, so that new vulnerabilities might be found. From the fuzzing tests we performed on the three programs—"palette", "std" and "json", we indeed found some vulnerabilities in the programs "palette" and "json". In the program "std", we did not find any vulnerable inputs. This does not mean there is no potential vulnerable input in "std". First, not all characters were put into the

vector for testing “std”. And also, the program was not tested with all possible combinations of characters in the vector. There is another thing needs to be noticed: the inputs space in the program “std” did not limit the string length, which implicitly increased fuzzing load, so that more resources were required to test the program. The three instantiations gave enough hints that large applications can be tested in the same way to find potential vulnerabilities.

6.2 Issues and future work

Yet there are still some unresolved issues that we did not concern about in the prototype of the grid fuzzing system, which include:

First, anyone in the network connecting to the ZooKeeper can modify information in ZooKeeper, which makes tampering ZooKeeper possible. If attacker can connect to the Zookeeper or behave like the remote resource but modify code on the resource side, the grid fuzzing system may not get fuzzing results. In the future, ZooKeeper can be configured to allow access control, so that malicious requests can be rejected, avoiding some security issues. Yet, we have to consider balancing making use of more resources and security of ZooKeeper.

The ZooKeeper was not configured to work in replicated mode in the thesis, which means that ZooKeeper’s failure would incur fatal consequence, because back-up servers are not available to support the fuzzing service. In the future, replicated mode of ZooKeeper can be configured to make the system tolerant failure.

As the grid fuzzing system is developed in Ubuntu and tested in the local area network (LAN) environment, but not in the wide area network (WAN) environment, the system’s compatibility is not guaranteed. It is quite likely that resources in another network have problems to obtain fuzzing tasks. For addressing this issue, the system needs more experiment in wide area networks to find out those elements that affect fuzzing. By overcoming these problems, they grid fuzzing system will be more compatible.

Additionally, in the future, we can improve the broker by fuzzing other applications (e.g., web applications) to exploit more types of vulnerabilities, enhancing system’s scalability and compatibility. Also, the broker may try to analyze the fuzzing feedbacks from ZooKeeper, make systematic vulnerability inspection in applications, and then provides security recommendations to clients.

Bibliography

[1] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In Proceedings of SOSP'05, October 23-26, 2005, Brighton, United Kingdom.

[2] Nuno Neves, Joao Antunes, Miguel Correia, Paulo Verissimo. Using Attack Injection to Discover New Vulnerabilities. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), pages 457-466, June 2006.

[3] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Jonida Silva Fraga. DepSpace: A Byzantine Fault-Tolerant Coordination Service. In Proceedings of EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

[4] The EDUCAUSE Learning Initiative. 7 Things You Should Know About Grid Computing (ID: ELI7010). http://www.educause.edu/ELI/7ThingsYouShould KnowAboutGridC/156813. January 2006.

[5] Fabio Favarim, Jonida Silva Fraga, Lau Cheuk Lung, and Miguel Correia. GRIDTS:A New Approach for Fault-Tolerant Scheduling in Grid Computing. In Proceedings of the 6th IEEE International Symposium on Network Computing and Applications (NCA), pages 187-194, July 2007.

[6] John Wack, Miles Tracy, and Murugiah Souppaya. Guideline on Network Security Testing. NIST Special Publication 800-42, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, October 2003.

[7] Samuel T. Redwine, Jr. and Noopur Davis. Processes to Produce Secure Software. Volume I: Software Process Subgroup of the Task Force on Security across the Software Development Lifecycle. National Cyber Security Summit, March 2004.

[8] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In Proceedings of the USENIX Annual Technical Conference, 2010.

[9] Dave Aitel. An Introduction to SPIKE, the Fuzzer Creation Kit. http://www.docstoc.com/docs/2687423/An-Introduction-to-SPIKE-the-Fuzzer-Creation-Kit, November 2008.

[10] Alysson Neves Bessani. DepSpace - A Byzantine Fault-Tolerant Coordination Service. http://www.navigators.di.fc.ul.pt/software/depspace/, February 2008.

[11] Tom White. Hadoop: The Definitive Guide, O' Reilly, June 2009.

[12] Apache Software Foundation. Apache Hadoop Project. http://hadoop.apache.org/zookeeper, May 2010.

[13] Brian J. Gough. An Introduction to GCC. Network Theory Limited, United Kingdom. Issue 16-4, August 2004.

[14] Apache Tomcat PMC members and Committers. Apache Tomcat Project. http://tomcat.apache.org, August 2010.

[15] Open Web Application Security Project. Cross-site scripting. http://en.wikipedia.org/wiki/Cross-site_scripting, October 2010.

[16] Open Web Application Security Project. Identify attack surface. http://www.owasp.org/index.php/Identify_attack_surface, May 2006

[17] Help network security. 1,800 Office bugs discovered by Microsoft's "fuzzing botnet". http://www.net-security.org/secworld.php?id=9092. April 2010.

[18] GRID Computing. http://voneural.na.infn.it/grid_comp.html, 2010.

[19] VMware. http://www.vmware.com

[20] George Notaras. Netcat – a couple of useful examples. http://www.g-loaded.eu/2006/11/06/netcat-a-couple-of-useful-examples. G-Loaded Journal, November 2006.

[21] Oracle and its affiliates. Reading Directly from a URL. http://download.oracle.com/javase/tutorial/networking/urls/readingURL.html, the Java tutorial, 2010.

[22] ASP Foundation. Top 10 2010-Main. http://www.owasp.org/index.php/ Top_10_2010-Main. April 2010.

[23] Ian Foster. What is grid computing? http://searchdatacenter.techtarget.com, Data center outsourcing, colocation and cloud computing, Data Center Hosted Services, 2010.

[24] ZooKeeper Overview. http://hadoop.apache.org/zookeeper/docs/r3.3.1/ zookeeperOver.html. May 2010.

[25] The Apache Software Foundation. Apache HTTP server project.

http://apache.org. 2010.

[26] Introducing Json. http://www.json.org.

[27] JSON in JavaScript. http://www.json.org/js.html.

[28] Mark Adler, etc. PNG (Portable Network Graphics) Specification, Version 1.2. Glenn Randers-Pehrson, 1999.

[29] Luis Ferreira, et al. Introduction to Grid Computing with Globus. IBM International Technical Support Organization, September 2003.

[30] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL Injection Attacks and Countermeasures. In Proceedings of the IEEE International Symposium on Secure Software Engineering, March 2006.

[31] William C. Hetzel, Bill Hetzel. The Complete Guide to Software Testing. 2nd edition, Wiley, 1993.

[32] William Perry. Effective methods for software testing. Wiley, 2006.

[33] John D. McGregor, David A. Sykes. A practical guide to testing object-oriented software. Addison-Wesley, 2001.

[34] Roger S. Pressman. Software Engineering. Chapter 18, software testing strategies. McGraw-Hill, April 2004.

[35] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. Information and Software Technology, pages 957-976, June 2009.

[36] Potter, B. Software security testing. Security & Privacy, IEEE, pages 81 – 85. Sept.-Oct 2004.

[37] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On Non-Functional Requirements in Software Engineering. Lecture Notes in Computer Science, pages 363-379, 2009.

[38] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro and Paulo Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. Lecture Notes in Computer Science, pages 61-86, 2003.

[39] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky and Dan Werthimer. SETI@home: an experiment in public-resource computing. Communications of

the ACM, Volume 45. November 2002.

[40] SETI@home. http://setiathome.berkeley.edu/.

[41] Grid Computing. http://en.wikipedia.org/wiki/Grid_computing. October 2010.

[42] DAME (DAta Mining & Exploration). GRID Computing. http://voneural.na.infn.it/grid_comp.html.

[43] Fuzz testing. http://en.wikipedia.org/wiki/Fuzz_testing. November 2010.

[44] Michael Sutton, Adam Greene, Pedram Amini. Fuzzing: Brute Force Vulnerability Discovery. 2007.

[45] AT&T Labs—Research. Black-Box Testing. John Wiley & Sons, January 2002.

[46] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 33(12):32–44, December 1990.

[47] Neil Chou, Robert Ledesma, Yuka Teraguchi, JohnC. Mitchell. Client-side defense against web-based identity theft. Computer Science Department, Stanford University, Stanford CA94305.

[48] Powerfuzzer. http://powerfuzzer.com.

[49] OWASP JBroFuzz. http://www.owasp.org/index.php/OWASP_JBroFuzz_Tutorial. September 2010.

[50] Inter-process communication. http://en.wikipedia.org/wiki/Inter-process _communication. October 2010.

[51] Andrew Watt. Beginning Regular Expressions. Wrox, 1st edition, February 2005.

[52] Computer cluster. http://en.wikipedia.org/wiki/Computer_cluster. October 2010.

[53] Zhendong Su, Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. POPL '06 January, 2006, Charleston, South Carolina, USA.

[54] Oehlert, P.. Violating assumptions with fuzzing. Security & Privacy, IEEE. March-April 2005.

[55] Wagner, D., Dean, R.. Intrusion detection via static analysis. Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on.

[56] Michael Eddington. Demystifying Fuzzers. January 2009.

[57] McAfee. Buffer Overflow, Exploits: The Why and How. McAfee System Protection Solutions. April 2005.

[58] John Bellardo, Stefan Savage. 802.11 denial-of-service attacks: real vulnerabilities and practical solutions. Proceeding SSYM'03 Proceedings of the 12th conference on USENIX Security Symposium - Volume 12, 2003.

[59] Data segment. http://en.wikipedia.org/wiki/Data_segment#BSS. November 2010.

[60] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers. Chapter 5, page 106-134: Concurrency and Race Conditions. 3rd edition, O' Reilly, January 2005.

[61] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. ACM Transactions on Embedded Computing Systems. Volume 4 Issue 4, November 2005.

[62] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan "noir" Eren, Neel Mehta, and Riley Hassell. The Shellcoder's Handbook: Discovering and Exploiting Security Holes. Wiley, April 2004.

[63] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time Detection of Heap-based Overflows. Proceedings of the 17th Large Installation Systems Administration Conference. October 26–31, 2003.

[64] Kyung-Suk Lhee, Steve J. Chapin. Software: Practice and Experience. Buffer overflow and format string overflow vulnerabilities, Volume 33, Issue 5, pages 423–460, April 2003.