ABSTRACT

Title of Thesis: CHECKING FOR APPLICATION VULNERABILITIES USING FAULT INJECTION

MELODY DJAM
Master of Science, July 2005
Reliability Engineering

Thesis Directed By: Dr. Michel Cukier
Center for Risk and Reliability,
Department of Mechanical Engineering

This thesis introduces a fault injector, called "Pulad", specifically developed for finding application vulnerabilities. Most previous approaches for finding application vulnerabilities involved static verification methods. With these methods, the source code is not executed. Since vulnerabilities can only be revealed when they are exploited, the use of a dynamic verification method, executing the source code, seems needed. The main two dynamic verification areas are software testing and fault injection. This thesis focuses on fault injection.

Pulad, the fault injector described in this thesis consists of two main parts called the "collector" and the "fault injector". The goal of the collector is to record all the environment-application interactions when the application is running. These interactions focusing on the environment files are then analyzed and the following fields are uploaded into a database including the file name, file extension, file size, file directory, number of times the file was used, file permission (includes symbolic link and ownership) and number of times an error occurred. The fault injector allows injecting faults either using a graphical user interface (GUI) or directly through a text file. The faults in the files include the file name, the directory name, the execution path, the library path, the file existence, the file ownership, the file permission, etc. For each of the faults, the specific type of fault needs to be indicated. Moreover, the interaction points where the faults should be injected are also provided by the user.

CHECKING FOR APPLICATION VULNERABILITIES

USING FAULT INJECTION

by

Melody Djam

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2005

Advisory Committee:

Dr. Michel Cukier, Chair
Dr. Ali Mosleh
Dr. Carol Smidts

Dedication

To my parents, Azar Manavizadeh and Davoud Djam

My sister, Maryam Djam

And my advisor Dr. Michel Cukier

Acknowledgements

It has been a wonderful experience for the past two years to work with my advisor Dr. Michel Cukier. He has been a constant source of encouragement and inspiration during various tough times and helped me going trough every step of my research. Without his directions and support, this thesis could not have been possible.

My greatest debt goes to my parents, my sister and my friends who have been encouraged and supported me all the way since the beginning of my studies. They offered me unconditional love and support throughout my education.

I would like to thank the Shrada Upadhyay and Rohit Krishna who helped me implementing Pulad.

Finally a word of thanks also goes to my laboratory mates Anil Sharma and Susmit Panjwani. They have been very cooperative and enlivened the work environment in the laboratory.

To each of the above, I extend my deepest appreciation

Table of Contents

Dedication ii
Acknowledgements iii
Table of Contents iv
List of Tables vi
List of Figures vii
Chapter 1: Introduction 1
Chapter 2: Approaches for Checking for Application Vulnerabilities 4
Introduction 4
Terminology 4
Static and Dynamic Verification Methods 5
Limitations of Static Analysis 7
Conclusions 10
Chapter 3: Checking for Application Vulnerabilities using Fault Injection 11
Introduction 11
Fault Injection using Environment Perturbation 11
Revised Approach for Conducting Fault Injection Using Environment Perturbation 16
Conclusions 20
Chapter 4: Collecting Environment-Application Interactions: Design and Implementation 21
Introduction 21
Overview of the Collector Architecture 21
Detailed Design of the Collector 23
Types of Environment-Application Interaction Files 31
Database Tables 32
Functional Descriptions 35
Sequence Diagrams 36

How to Run Pulad 37

Conclusions 39

Chapter 5: Fault Injection for Finding Application Vulnerabilities: Design and Implementation 40

Introduction 40

Overview of the Fault Injector Architecture 40

Fault Injection techniques for Fault Injector 43

Detailed Design of the Fault Injector 45

Functional Description 50

Conclusions 53

Chapter 6: Validation of Pulad 54

Introduction 54

Selected Testing Techniques 55

Examples of Applications 57

Testing Results 58

Conclusions 69

Chapter 7: Conclusion and Future Work 71

Appendix A: Code of the tracking system written in C: 74

List of Tables

Table 1: Vulnerabilities Identified by Static Analysis Tools 8

Table 2: Number of False Positives and False Negatives 9

Table 3: Buffer Overflows Identified by Static Analysis and the Evolved Strategy 10

Table 4: Sub-categories of Indirect Faults 15

Table 5: Sub-categories of Direct Faults 17

Table 6: Environment Faults Considered in this Thesis. 20

Table 7: Fault Injection Considered in this Thesis. 21
Table 8: Example of Input File 43

Table 9: Sorting of Files Based on Interaction Times 44

List of Figures

Figure1: Collector Architecture Overview 22

Figure2: Collector Sequence Diagrams 37

Figure 3: Fault Injector Architecture Overview 41

Figure4: Fault Injector Sequence Diagrams 52

Chapter 1: Introduction

Everyday new vulnerabilities are discovered in operating systems, services and applications. These vulnerabilities would not be such an issue would the black-hat community or attackers not be so active. In a recent paper [Pan05], it was shown that 2 computers left with about 25 vulnerabilities open would be attacked by 760 different attackers (each attacker being associated with a source IP address) in just 48 days of data collection. Since vulnerabilities left open are so quickly exploited, the issue of finding vulnerabilities is critical.

Vulnerabilities can be classified into host, network and application vulnerabilities based on their location. Tools exist for checking for host and network vulnerabilities. Since these vulnerabilities are located in operating systems and services used by a large community, vulnerabilities have a high probability to be discovered. Tools can then be used to check if the operating system or services used contain the vulnerabilities. However, most of the code is not contained in operating systems or services but in applications [Vie02a]. Since most applications are used by a smaller community, some vulnerabilities have a lower probability of being found. Moreover, few tools exist for checking for these vulnerabilities. In fact, most existing tools rely on the source code of the application and analyze statically if vulnerabilities are present. This method only indicates potential vulnerabilities and misses some vulnerabilities since the code is not executed. Finding an approach for checking for application vulnerabilities based on a dynamic approach that executes the application

code is an important issue. This thesis tackles this issue applying fault injection for finding application vulnerabilities.

This thesis is structured as follows. In Chapter 2, we first introduce the terminology we will use through the entire thesis. We then review the most well-known static and dynamic verification methods applied to finding application vulnerabilities. We then identify the limitations of using only static verification methods.

In Chapter 3 we focus on one specific dynamic verification method, fault injection. [Gho98] perturbed the internal state of the executing application and [Du00] perturbed the environmental state. When neither approach has led to a tool that can automatically check for application vulnerabilities, the fault injector described in this thesis is based on a revised version of the framework developed by [Du00] to conduct fault injection. The first part of Chapter 3 will review the concepts introduced by [Du00]. The second part of Chapter 3 will detail the concepts used for the fault injector presented in this thesis and the issues associated with the proposed approach.

We described in Chapter 4 and 5 the tool that we have developed. Our fault injector is called “Pulad” and consists of two main components. The first main component focuses on the collection of environment-application interactions related to files when the application is executed and is called the “collector”. The second main component focuses on the injection of faults and is called the “fault injector”. The collector will be described in Chapter 4. The fault injector will be described in Chapter 5. In both chapters, we will describe the collector and fault injector at a high level, then provide design and implementation details. A functional description, sequence diagrams and details on how to execute the collector will also be provided.

In Chapter 6, we focus on the validation of Pulad based on software testing. Three applications were used and developed. Three different testing techniques were used to ensure the proper behavior of Pulad.

Finally, in Chapter 7, we conclude this thesis and list issues and future work.

Chapter 2: Approaches for Checking for Application Vulnerabilities

Introduction

In this chapter, we first introduce the terminology we will use through the entire thesis. We then review the most well-known static and dynamic verification methods applied to finding application vulnerabilities. We then identify the limitations of using only static verification methods by comparing the outcome obtained by some popular static analysis tools with a revised testing strategy targeting buffer overflows. This example illustrates the need for developing dynamic verification methods for finding application vulnerabilities. Instead of software testing, this thesis focuses on another important dynamic verification method: fault injection.

Terminology

Over the years, software in computing systems has become significantly larger and more complex. This increase of size and complexity leads to an increase of the number of faults or bugs present in the software. Most faults might lead to software failures when the fault is activated. However, some of these faults might also be targeted by malicious users (attackers). Indeed, some of these faults might lead to system compromises when an attacker exploits these faults. More precisely, this special type of fault is called a *(security) vulnerability*. A vulnerability is a weakness

of the computing system that can be exploited by an attacker. An *exploit* of a vulnerability can lead to an *intrusion* in the computing system by the attacker. Building a secure computing system thus requires to find and remove vulnerabilities. Vulnerabilities are usually classified into host vulnerabilities, network vulnerabilities, and application vulnerabilities. Host vulnerabilities are linked to potential attacks from insiders and lead to potential theft and abuse of privilege (i.e., improper use of authorized operations). Network vulnerabilities are linked to potential attacks from outsiders and lead to potential theft of privilege (i.e., unauthorized increase in privilege). Like host vulnerabilities, application vulnerabilities allow a theft of privilege and an abuse of privilege. The definition of the concepts related to security presented in this section are taken from [MAFTIA00] and [MAFTIA01].

This thesis focuses on application vulnerabilities. Since most of the currently developed code belongs to the application category, application vulnerabilities are of paramount importance. However, among all possible application vulnerabilities, so far the security community has mainly focused on buffer overflows (considered as the vulnerability of the decade [Cow99]) and race conditions [Vie02a]. Like for any other fault, finding (application) vulnerabilities can be done by using either static verification methods (without software execution) or dynamic verification methods (with software execution).

Static and Dynamic Verification Methods

The main static verification methods are formal methods and static analysis. The advantage of the use of formal methods is the precision associated with the approach.

The disadvantages are the difficulty in specifying the requirements and the system and the development of techniques for checking the requirements specification against system specification. Static analysis is mainly based on code analysis to identify potential vulnerabilities (i.e., vulnerabilities that can potentially be exploited by attackers). Static analysis is a lightweight approach to find with high efficiency potential vulnerabilities. However, since the software is not executed, only potential vulnerabilities can be identified. Moreover, vulnerabilities might also be missed when applying static analysis. Since many tools are currently apply static analysis approaches, we will examine in detail later in this chapter the efficiency of some of the most popular tools.

Dynamic verification methods include fault injection and software testing. Two recent approaches applying fault injection both simulated the incoming attacks. [Gho98] perturbed the internal state of the executing application and [Du00] perturbed the environmental state. However, none of these approaches can be easily ported into a tool checking for generic application vulnerabilities. Among the software testing strategies, penetration testing is the main strategy where assumptions on possible vulnerabilities are made by accessing, for example, documents on the application and its environment. Each of these assumptions is then verified by testing [Dow01, Dan99]. [McG98] points out that penetration testing occurs too late in the software development process, just before software is released. Moreover, there is no rigorous approach associated with penetration testing. Besides penetration testing, very few software testing strategies have been applied in the context of security. Of those few, the PROTOS project [PROTOS] at the University of Oulu, Finland,

focuses on testing the security of protocol implementations. The researchers created several test suites, available on the project web site, that identified vulnerabilities in most of the tested products, leading even to a CERT advisory (CA-2002-03) for vulnerabilities found in many implementations of the Simple Network Management Protocol (SNMP). As part of the PROTOS project, [Kak00] developed a method of vulnerability analysis through syntax testing. Moreover, at Purdue University, [Asl96] developed a taxonomy of security faults and linked the faults with software testing strategies. Faults are classified into synchronization errors, condition validation errors, configuration errors, and environment faults. The surveyed testing strategies include symbolic testing, path analysis testing, functional testing, syntax testing, and mutation testing. Finally, [Vie02a] claims that black box testing is not very effective in the context of security. White box testing however is claimed to be much more effective.

Limitations of Static Analysis

As previously mentioned, static analysis is a well-known static method that is mainly based on code analysis to identify potential vulnerabilities (i.e., vulnerabilities that can potentially be exploited by attackers). However, static analysis cannot identify the environmental conditions needed for some vulnerabilities to be exploited. Therefore, a number of false positives (i.e., identified vulnerabilities that cannot be exploited by an attacker) can be expected. Tools applying static analysis include Flawfinder [Flawfinder], RATS [RATS], and ITS4 [Vie02b]. RATS, ITS4, and Flawfinder use a built-in database of vulnerabilities. For example, Flawfinder uses a built-in database

of C/C++ functions with well-known vulnerabilities. On the other hand, CQUAL uses constraint-based type inference. To analyze a program, CQUAL traverses the program's abstract syntax tree and generates a series of constraints that capture the relations between type qualifiers. Since the exploit of a vulnerability might depend on environmental conditions that cannot be identified by static verification methods, dynamic verification methods complement static analysis. The following example illustrates the limitations of exclusively using static analysis. We applied Flawfinder, RATS, and ITS4 on the "mingetty.c" benchmark which is described as "a small, efficient, console-only getty for Linux that opens a tty port, prompts for a login name and invokes the /bin/login command" [mingetty]. We obtained the results given in Table 1.

Vulnerability Type / Tool	FLAWFINDER	RATS	ITS4
Buffer Overflow	21	10	6
Format String	2	2	14
Race Condition	2	0	10

Table 1: Vulnerabilities Identified by Static Analysis Tools

The 21 buffer overflow vulnerabilities identified by Flawfinder include the 10 and 6 buffer overflow vulnerabilities found respectively by RATS and ITS4. An evolved version of robust worst-case boundary value analysis testing was applied independently[1] to find buffer overflows and found a total of 13 buffer overflows. The vulnerabilities found by robust worst-case boundary value analysis were then

1 The development and application of the evolved testing technique was conducted by Avik Sinha and Ming Li under the supervision of Dr. Carol Smidts.

compared with the vulnerabilities found by the static analysis tools. The number of false positives and the number of vulnerabilities missed by the static analysis tools (i.e. false negatives) are shown in Table 2.

Static Analysis Tool	Number of False Positives	Number of False Negatives
ITS4	3 (50%)	10 (77%)
RATS	7 (70%)	10 (77%)
Flawfinder	8 (38%)	0 (0%)

Table 2: Number of False Positives and False Negatives

By inspection it was found that some of the vulnerabilities initially identified by static analysis tools were not actual buffer overflows since the application contained bound checks to protect against range violations. Flawfinder includes the 13 actual buffer overflow vulnerabilities among the 21 identified vulnerabilities. In addition Table 2 shows that not all vulnerabilities could be identified using ITS4 and RATS, because mingetty.c contains some user- built functions, which are not included in the vulnerability definition libraries of the tools. This example shows that testing can be used in conjunction with static analysis to remove false positives. This example also shows that static analysis tools are not always effective at finding buffer overflows caused by user-defined functions.

In another example, we applied the same static analysis tools (i.e., Flawfinder, RATS, and ITS4) on a tracking system, Tracker.c (presented in Appendix A), written in C that contains one buffer overflow vulnerability. Tracker.c is a projectile tracking system that has user defined library functions that implement data input and processing. The results of static analysis showed that no buffer overflow was found

by any of the static tools. We[2] then developed a suite of test cases based on the evolved testing strategy. The result showed that a buffer overflow was present in the code. This example confirms the previously mentioned result, i.e. that static analysis tools are not always effective at finding buffer overflows caused by user-defined functions (see Table 3). Based on the limitations of static analysis and the inherent ability of dynamic approaches to reproduce the execution environment, dynamic approaches should be used in complement to static analysis.

Static Analysis Tool	Number of False Positives	Number of False Negatives
ITS4	0	1
RATS	0	1
Flawfinder	0	1

Table 3: Buffer Overflows Identified by Static Analysis and the Evolved Strategy

Conclusions

This chapter reviews different static and dynamic verification methods for finding application vulnerabilities. The pros and cons of different approaches are described. In particular, the limitations associated with using only static verification methods are detailed. The conclusion of this chapter is the use of dynamic verification approaches combined with static analysis. In the next chapter we detail a dynamic verification method based on fault injection.

---

2 The development and application of the evolved testing technique was conducted by Avik Sinha and Ming Li under the supervision of Dr. Carol Smidts.

Chapter 3: Checking for Application Vulnerabilities using Fault Injection

Introduction

In Chapter 2, we identified the limitations of the sole use of static verification methods for finding application vulnerabilities and the usefulness of combining static and dynamic verification methods. In this chapter we focus on one specific dynamic verification method, fault injection. In the previous chapter, we stated that the two recent approaches applying fault injection both simulated the incoming attacks. [Gho98] perturbed the internal state of the executing application and [Du00] perturbed the environmental state. When neither approach has led to a tool that can automatically check for application vulnerabilities, the fault injector described in this thesis is based on a revised version of the framework developed by [Du00] to conduct fault injection. The first part of this chapter will review the concepts introduced by [Du00]. The second part of the chapter will detail the concepts used for the fault injector presented in this thesis and the issues associated with the proposed approach.

Fault Injection using Environment Perturbation

Like [Du00], we assume that a "system" combines an "application" and its "environment". Based on this definition, all code that is not part of the application would be part of the environment. The range of the environment can be reduced by

only defining as “environment” the portions of the code that have a direct or indirect coupling with the application code. The use of common resources (e.g., files, network components) or global variables are examples of such couplings. [Gar96] and [Krs98] empirically demonstrate that the environment plays a significant role in triggering vulnerabilities that lead to security policy violations.

We define a “secure” program as a program that tolerates environment perturbations without any security policy violation. If we now consider environment perturbations as faults, we then consider a secure system as a fault-tolerant system able to tolerate faults in the environment. Fault injection can be defined as “the deliberate insertion of faults into an operational system to determine its response” [Cla 95]. In the approach introduced by [Du00], faults are injected in the application environment and thus perturbating the environment. The perturbation then might lead to a security violation. If it does not lead to a security violation, then the application is considered secure.

The terminology introduced by [Du00] defines:

- Internal entity: any element in the application’s code and data space.
- Internal state: a state that consists of the status of the internal entities.
- Environment entity: any element that is external to an application’s code and data space.
- Environment state: a state that consists of the status of the environment entities.

Examples include: a variable in an application (i.e., internal entity), the value of the

variable (i.e., internal state), files and network (i.e., environment entities), the permission or ownership of a file (i.e., environment state). The shared nature of the environment entity differentiates internal entities from environment entities. An environment entity is not only accessed and changed by an application. Other objects also can access and change an environment entity. This is not the case with internal entities that only applications can access and modify.

Environment faults usually affect an application in two ways [Du00]. An application can receive inputs from its environment. In that case the associated environment faults are faults in the input. The input is included in an internal entity of the application. The fault then propagates through the application via the internal entities. A security violation might occur if the application is not able to correctly handle the faults. When the direct reason of this violation seems to be due to the faults in the internal entities, the real reason is the propagation of environment faults. The environment thus indirectly causes a security violation via the internal entities. These faults are called *indirect environment faults*. For example, assume that an application receives its input from the network. Any fault in the network message is included in the internal entity. When the application copies this message into a buffer without checking the buffer's boundaries, a security violation occurs. Indirect environment faults can be divided into the following sub-categories according to their origin:

- user input
- environment variable
- file system input

- network input
- process input

The different sub-categories are summarized in the following table [Du98].

Indirect Fault	Semantic Attribute	Description
User Input	file name	Name of a file
	directory	Name of a directory
	command	Name of a command executed in the application
Environment Variable	file name	Name of a file
	directory	Name of a directory
	execution path	List of paths used to search executable files or commands
	library path	List of paths used to search libraries
	permission mask	A mask which decides default permission of a newly created file
File System Input	file content	Content of a file
	file name	Name of a file
	directory	Name of a directory
	file extension	Special string that represents that type of files
Network Input	IP address	Representation of IP address
	packet	Packet
	host name	String that represents the name of host
	DNS reply	Reply from DNS server
Process Input	message	Message sent from one process to another

Table 4: Sub-categories of Indirect Faults

The second way environment faults affect an application is when the fault remains within the environment entity and when the application interacts with the environment without correctly handling these faults. In that case, a security violation occurs. Environment faults are then the direct cause of the security violation and the medium for environment faults is the environment entity itself. These faults are called *direct environment faults*. For example, when an application needs to execute a file, the owner of the file might be the owner of the application or some malicious user. In case the application does not check who the owner of the file is, some arbitrary code

might get executed leading to a security violation. Indirect environment faults can be divided into the following sub-categories:

- file system
- process
- network

The different sub-categories are summarized in the following table.

Direct Fault	Attribute	Description
File System	file existence	File does or does not exist
	file ownership	Owner of the file
	file permission	Access permission for different users
	symbolic link	File is a symbolic link to another file
	file content invariance	File can or cannot be modified during the execution of the application
	file name invariance	File name can or cannot be modified during the execution of the application
	working directory	Directory where the application is invoked
Network	message authenticity	Message is genuine or is spoofed by other people
	protocol	Message from network does or does not comply with underlying protocol
	status of socket	Socket is or is not shared with another process
	availability of service	Network service is or is not available
	trustability of entity	Entity at the other end of network is or is not trusted
Process	message authenticity	Message is genuine or is spoofed by other people
	trustability process	Process with which the application is communicating is or is not trusted
	availability of service	Service that the application requested is or is not available
	protocol	Message from another process does or does not comply with underlying protocol

Table 5: Sub-categories of Direct Faults

Revised Approach for Conducting Fault Injection Using Environment Perturbation

The fault injector we have developed is based on a similar theoretical framework. However, several significant differences exist between the framework proposed by [Du00] and the one we have developed. First, we only consider environment faults through files and not faults through the network or processes. Second, we also no longer maintain the distinction between indirect and direct environment faults. We now detail the reasons for making these changes.

[Krs98] analyzed a security vulnerability database consisting of around 195 application vulnerabilities from different operating systems such as Windows-NT, Solaris, HP-UX, and Linux. Among these vulnerabilities, 142 could be used to be classified as indirect or direct environment faults. 57% of the vulnerabilities could be identified as indirect environment faults, 34% as direct environment faults, and 9% could not be categorized as neither indirect nor direct environment faults. Among the indirect environment faults, 90% were linked to files, 10% to network inputs and 0% to process inputs. Among the direct environment faults, 87% were linked to files, 10% to the network and 2% to processes. The first reason for focusing on environment faults linked to files is the significant number of vulnerabilities associated with files. Moreover, the approach proposed by [Du00] to find application vulnerabilities requires: a) to have access to the source code of the application, and b) to manually analyze the source code to find specific cases that would lead to a security violation. A fault would then be injected to verify that indeed the fault leads

to a security violation. As previously mentioned, the automation of such a process would be very complex. The most complex analysis of a security violation involves issues related to "trust" (i.e., trustability of entity, trustability process). Both cases involve the network and processes. So, in order to reach some automation and avoid systematic manual analysis, we will no longer consider issues related to the network and processes. Since network and process environment faults count for about 10% of the cases, this condition for reaching some level of automation only discards about 10% of the environment faults leading to a security violation.

As already mentioned, the method proposed by [Du00] requires the analysis of the source code of the application to identify potential security violations when specific faults are injected. This approach leads to a very small number of faults that need to be injected but requires a complex manual analysis before fault injection and is based on the source code. We do not believe that these assumptions are reasonable for most applications. Source code is not often made available, Moreover, in practice, rare will be the cases when an organization agrees that some programmers spend large amounts of time analyzing manually an application source code to identify the faults to be injected that would lead to a security violation. Therefore, we decided to take the "black-box" approach, assuming that the source code would not have been made available. Moreover, our goal is also to move from manual analysis to some automation. The direct consequence is that we no longer will be able to identify precisely which faults will lead to a security violation. A significant higher number of faults will need to be injected. And for these faults, some analysis is needed after fault injection to check if the fault led to a security violation. Mainly because we opted for

the black-box approach, the distinction between indirect and direct environment faults becomes less relevant. The following table presents the environment faults that will be considered from now.

Fault	Attribute	Description
User Input	file name	Name of a file
	directory	Name of a directory
	command	Name of a command executed in the application
Environment Variable	file name	Name of a file
	directory	Name of a directory
	execution path	List of paths used to search executable files or commands
	library path	List of paths used to search libraries
	permission mask	A mask which decides default permission of a newly created file
File System Input	file content	Content of a file
	file name	Name of a file
	directory	Name of a directory
	file extension	Special string that represents that type of files
File System	file existence	File does or does not exist
	file ownership	Owner of the file
	file permission	Access permission for different users
	symbolic link	File is a symbolic link to another file
	file content invariance	File can or cannot be modified during the execution of the application
	file name invariance	File name can or cannot be modified during the execution of the application
	working directory	Directory where the application is invoked

Table 6: Environment Faults Considered in this Thesis

The goal of the fault model proposed by [Du00] is to allow fault injection to be conducted at the environment-application interaction level to try to "emulate what a "real" attacker does". The claim made by the authors is that "since most of the vulnerability databases record the way attackers exploit a vulnerability, we transform these exploits to environment faults to be injected with little analysis on those records

thereby narrowing the semantic gap between faults injected at the interaction level and faults that really occur during the intended use of the system." This goal is relevant when conducting an extensive manual analysis of the code to identify potential security violations. Since we are taking the black-box approach, we have less insight on the application. Therefore, out goal should not to emulate "real" attacks but rather perturb in many different ways the environment to check the cases when a security violation appears. This approach has also the advantage of not assuming what the attacker might do and therefore also includes original attacks that we would not have thought of. Our approach can thus been seen as checking if building blocks linked to the files that attackers could use to develop attacks might lead or not to a security violation.

The following table lists the different possible fault injections for the attributes that we have identified.

Entity	Attribute	Fault Injection
User Input	file name	Change length, use relative path, use absolute path, insert special characters such as "..", "/" in the name
	directory	Change length, use relative path, use absolute path, insert special characters such as "..", "/" in the name
	command	Change length, use relative path, use absolute path, insert special characters such as "|", "&", ">" or new line in the command
Environment Variable	file name	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name
	directory	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name
	execution path	Change length, rearrange order of path, insert a untrusted path, use incorrect path, use recursive path
	library path	Change length, rearrange order of path, insert a untrusted path, use incorrect path, use recursive path
	permission mask	Change mask to 0 so it will not mask any permission bit
	file name	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name

	directory	Change length, use relative path, use absolute path, use special characters such as "|", "&", ">" in the name
	file extension	Change to other file extensions like ".exe" in Windows system; change length of file extension
File System	file existence	Delete an existing file or make a non-existing file exist
	file ownership	Change ownership to the owner of the process, other normal users, or root
	file permission	Flip the permission bit
	symbolic link	If the file is a symbolic link, change the target it links to; if the file is not a symbolic link, change it to a symbolic link
	file content invariance	Modify file
	file name invariance	Change file name
	working directory	Start application in different directory

Table 7: Fault Injection Considered in this Thesis

Conclusions

In this chapter we described an approach for identifying application vulnerabilities applying fault injection using environment perturbation. We then revised some of the concepts of the described approach so that automation would be possible and having the application source code would not be required. Based on these new concepts, Chapter 4 and Chapter 5 introduce the tool we have developed for finding application vulnerabilities using fault injection.

Chapter 4: Collecting Environment-Application Interactions: Design and Implementation

Introduction

After having motivated the choice of applying a dynamic verification method for checking for application vulnerabilities using fault injection, we describe in Chapter 4 and 5 the tool that we have developed. Our fault injector is called “Pulad” which means “hard, strong, hard to break” in Persian. The story behind Pulad is deeply rooted in Persian culture. Pulad consists of two main components. The first main component focuses on the collection of environment-application interactions related to files when the application is executed and is called the “collector”. The second main component focuses on the injection of faults and is called the “fault injector”. The collector will be described in Chapter 4. The fault injector will be described in Chapter 5.

Overview of the Collector Architecture

For running the collector, the user only needs to provide the application’s execution command and the directory name of the specific application as inputs. The output of the collector is then stored in an Oracle database table. The following figure shows the overview of the collector architecture.

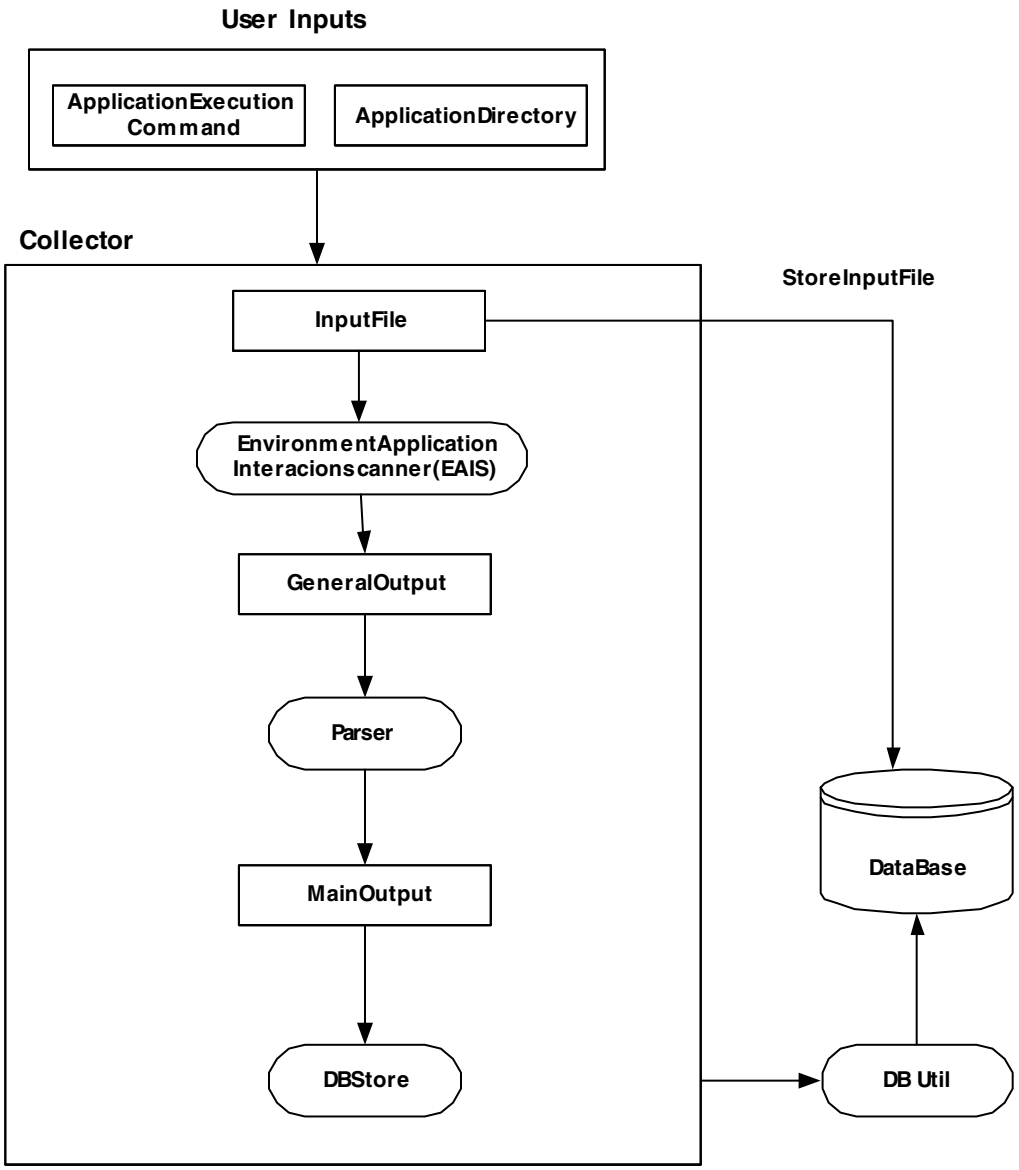

Figure1: Collector Architecture Overview

First, the user needs to provide the application execution command and the application directory name. This data is needed as input to the collector and is stored in the collector input file. The input file is then stored in the database. Using this information, the collector then runs the Environment-Application Interaction Scanner

(EAIS) module to capture all the environment-application interactions associated with files. The output of EAIS is stored in the in collector GeneralOutput file. The parser in the collector then parses the GeneralOutput file to remove all extra symbols generated by Java (Java will add some unnecessary data when executing the application), changes the time reference, and saves the output into the MainOutput file. DBStore then scans the MainOutput file and saves the information in the Oracle database through the DBUtil module (DBUtil connects the application with the Oracle database engine and ports data from DBStore to the Oracle database).

Detailed Design of the Collector

In this section we detail the different parts of the collector.

The Input File

The input file stores the user input provided to the collector. As already mentioned, the user inputs consist of the application execution command and the application directory name. The application execution command is the command that is used to run the specific application. For instance, the "ls" command in UNIX is a simple application execution command. The application directory name is the name of the directory where the application is located. For instance, the directory name for the same UNIX command, ls, is "/bin". The user provides this information in the following format: "ls /bin collector". Finally, as a third argument, the user needs to enter a keyword, "co" or "fi", referring either to running the collector component or the fault injector component (to be described in Chapter 5). These three inputs are

also saved in the Oracle database. The collector or fault injector can then retrieve these user inputs at anytime.

The EAIS Module

The Environmental Application Interaction Scanner (EAIS) module is a module that captures all environment-application interactions involving files. This module intercepts and records the system calls, which are called and received by the application. EAIS consists of a modified version of the tool called strace [strace]. This open source tool is a useful diagnostic, instructional, and debugging tool. It captures most interactions between the application and the environment. But it would not give all the information for the specific file system. For instance strace will not provide the ownership of the file that interacted with the application. Also, the times of the interactions recorded by strace (the first eighteen digit of each line) refer to the time that strace was built. We modified strace to get the ownership of the file. Also we modified strace to change the time reference of each interaction. The output format of EAIS is the identical to the one used by strace. We added an argument to show the ownership of each file system. The following is the pseudo-code of EAIS to capture the file ownership:

```
For each file interacting with the application, after the
interaction(while it is open)
{
 run the UNIX command “ls -al”;
 parse the output of “ls -al” to get the ownership;
```

save the ownership name in the output file as 3rd argument in the lines starting with "open";

}

The following is the output of "ls –al bash_logout".

```
-rw-r--r--    1 Root   oinstall       24 Dec 20 19:01 .bash_logout
```

The third argument shows the ownership of the file. In the above example,the owner is "root". So EAIS will trace the ls output, capture the ownership and save it into the GeneralOutput file as the third argument in of each line starting with "open". The command to run the EAIS module in the collector is as follow :

```
strace -ttt [-o GeneralOutput] [Application execution command]
```

This command runs the application and stores the output in the GenaralOutput file. The argument "–ttt" is used so that each environment-application interaction time is recorded. The "Application execution command" argument contains the command to execute the application provided by the user.

The GeneralOutput File

The GeneralOutput file contains the output of the EAIS module. This output contains some unnecessary symbols that Java added while running this module. The following shows the content of the GeneralOutput file when running the "ls" UNIX command:

```
1114466952.522425 execve("./ls", ["./ls"], [/* 37 vars */]) = 0
1114466952.522803 uname({sys="Linux", node="Redhat9", ...}) = 0
1114466952.523025 brk(0)                = 0x804a368
1114466952.523106 old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
1114466952.523206 open ("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT
```

```
(No such file or directory)
```

The Parser Module

To remove all unnecessary symbols added by Java while running EAIS and change the time reference (removing the first 10 digits) we developed the parser module. This module parses the GeneralOutput file and deletes all extra arguments that were generated by Java. The output of this module will be saved in the MainOutput file. For instance, in the GeneralOutput file, each file directory contains:

```
/usr/java/j2sdk1.4.2_06/jre/lib/i386/client/tls/i686
```

which is the directory where Java is running. The parser module removes these directories and stores the rest in the MainOutput file. As we can see from the GeneralOutput file presented in the previous section, each line that starts with “open”, “fstat” or “stat” contains the directory path that Java creates. The parser also cuts the first 10 digits of 16 digit long timestamps, removes the “.” in the time format, and removes the space after the timestamp. The parser will parse this output file based on the following pseudo-code.

```
Create file a =GeneralOutput file
While each line in a != Null
{
  int q=line.indexOf('(');
  b=line.substring(16,q);
  S_command=Take the command after b;
  If S_command =”open” then
  {
   take the first argument;
   trace all the directories that contains java path;
   omit the java path;
   take the first sixteen digits and take the first argument after
“open”;
   omit the first ten digits for each timestamp;
  }
  If S_command = ”close” then
  {
   take the fist sixteen digits;
   omit the first ten digits for each timestamp;
  }
```

```
  If S_command = ”fstat” or “stat” then
  {
   take the first argument;
   trace all the directory that contains Java path;
   omit the Java path;
   take the first sixteen digits before “fstat” or “stat”;
   omit the first ten digits for each timestamp;
  }
save in MainOutput file
}
```

The MainOutput File

The MainOutput file is the output of the parser module. In this file, each line contains the timestamp (first six digits), the interaction file name, followed by its arguments in parentheses and its return value. The arguments associated with the interaction files include the directory name, permission, file extension, file ownership and size of the file.

The following shows the MainOutput file when executing the UNIX “ls” command:

```
522425execve("./ls", ["./ls"], [/* 33 vars */]) = 0
522803uname({sys="Linux", node="Redhat9", ...}) = 0
523025brk(0)                                    = 0x804a368
523106old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
523206open("/etc/ld.so.preload", O_RDONLY, root) = -1 ENOENT (No
such file or directory)
```

As you we can see from this output, all the unnecessary directory names and associated symbols were removed by the parser module. The sixteen digit numbers were also removed from the GeneralOutput file to only keep the last six digits in the MainOutput file.

The DBStore Module

After having caught all the information related to the environment-application interaction files, we store it in database tables (so that we can retrieve the information

later). The information stored related to each files is:

- The file name, which contains the name of each environment-application interaction file.

  Each interaction file name can be found in the MainOutput file. This data is the first argument of the line that starts with ”open”. For instance from the following line, we can capture the name of the interaction file which is ls.so.cache: open("/etc/ld.so.cache", O_RDONLY, root) = 3

- The directory name where the file is located.

  The directory name can be found from the MainOutput file. Each directory name is the first argument of each line that starts with ”open”. In the previous example, the directory is “/etc/”.

- The owner of the file, who can make changes to the file.

  This field can be captured from the third argument of each line that starts with “open” in the MainOutput file. In the previous example, the owner is “root”.

- The file permissions, which are the permissions associated with the file.

  Permission of each file can be captured from the second argument of each line that starts with “open” in the MainOutput file. In the previous example, the permission is “O_RDONLY” which indicates that this file can just be opened to be read.

- Open time, which is the time when file started interacting with the application.

  This field consists of the first six digits of each line that start with “open” in MainOutput file.

- Close time, which is the time when the file stopped interacting with the application.

  The close time can be captured from the first six digits of the line that starts with the “close”. This line shows that the interaction file stopped interacting with the application.

- File size, which is the size of the file interacting with the application.

  The file size is the second argument of the line that starts with “fstat” or “fstat3”.

- File existence.

  The value assigned to the line that starts with “open” shows the existence of the file. If the number is 3, it means that file interacted with the application. If the number is -1 it means that the file could not be interact with the application. For instance the first line in the following shows that ld.so.preload file could not interact with the application .The second line shows that ld.so.cache file interacted with the application:

```
open("/etc/ld.so.preload", O_RDONLY, root) = -1
open("/etc/ld.so.cache", O_RDONLY, root) = 3
```

All this information can be obtained from the MainOutput file. For instance, from the following lines, we can get the interaction file name which is “libc.so.6”, the directory name where the file is located which is “/lib/tls/”, the file permission which is O_RDONLY and the owner of the file which is root. Also from the second line, we can get the size of the interaction file, which is 50025 bytes. The third line gives some information about the memory map of the file, while it was interacting with the

application. The fourth line, which is “close(3)=0”, indicates that the interaction file stopped interacting with the application.

```
open("/lib/tls/libc.so.6", O_RDONLY, root) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=50025, ...}) = 0
old_mmap(NULL, 50025, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)=0
```

The DBStore module takes the information from the MainOutput file and stores it into the specific table in database. The following pseudo-code shows how the information is captured from the MainOutput file:

```
Create file a= MainOutput file
While each line in a != Null
{
  q=Get the first world
  If start=”open” then
   {
   Get the start time;
   Store it into start time field in database;
   pass “(“;
   look for the first argument
    Store it into filename field in database;
    Store the directory into file directory field in database;
    Get file type from directory;
    Store it into file type field in database;
    Take the second argument;
    Store it into file permission field in database;
    Take the third argument;
     Store it into ownership field in database;
Go to next line;
   }
if q =”fstats” Then
 {
   Take the second argument;
   Store it into number of link filed in the database;
  }
   if q=”close” Then
   get the end time;
   store it into end time field in database;
   store it in to the start time and the ending time fields in the
database
 }
```

The algorithm ports the MainOutput file to a temporary file called “file a”. First it

checks if the line is not null. Then for each line, it looks at each line's first word. If the line begins with "open", it ports the start time, the first argument to the file's name and the directory to the database. Then it ports the second argument to the file permission's directory. Thereafter, it gets the third argument and ports it to the ownership field. If the line starts with "fstats", then the algorithm takes the second argument and ports it to the size field. If the line starts with "close", it ports the ending time to the database.

Types of Environment-Application Interaction Files

The environment-application interaction files can be categorized as follows:

Temporary files. These files are created and deleted while the application is running. Temporary files are located in the "/tmp" or "$temp" directories. These files are identified by searching the MainOutput file, if the first argument in parenthesis of each system call contains the path "/tmp/<file name>". If it does, the application is using that particular temporary file.

Environment files. All global environment variables that different programs use are located in the "/etc", "/etc/env.d", "/etc/profile.env" or "/etc/profile.ed" directories. These files are identified by searching the MainOutput file, if the first argument in each system call contains one of the above environment directories. If it does, the application is using that particular environment variable.

Library files. Library files are global files and could be accessed by other applications. These files are located in the "/lib" directory. Therefore, we just need to check for this directory in the MainOutput file to identify if the application has

interacted with any library file.

Database Tables

We designed three tables in Oracle to store data. The three tables are the following.

File table. In this table the fields contain general information on each interaction file. The fields are as follow:

- File ID

This is the unique ID number that is assigned to each interaction. We use this ID number as a primary key of the table.

- Interaction file name (which is the primary key of this table)

Interaction file names are identified as the first argument after ”open” in each line in the MainOutput file. For instance, in

```
open("/lib/tls/libc.so.6", O_RDONLY, root) = 3
```

`libc.so.6` is the name of the interaction file.

- File type

The type of each interaction file can be found from the directory where the file is located. As previously mentioned, each type of file is located in a specific directory. For instance `/lib/tls/libc.so.6` shows that the file is located in the lib directory, indicating the file type is a library file.

- File size

The file size is captured from second argument in each line that starts with “fstat”. The size is indicated after “st_size” in the second argument. For instance the following line in the output of the MainOutput file shows the size of 50025 for

ld.so.cache file

```
open("/etc/ld.so.cache", O_RDONLY, root) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=50025, ...}) = 0
```

- File directory

The file directory name can be captured from the first argument of each line that starts with “open”. For instance line

```
open("/etc/ld.so.cache", O_RDONLY, root) = 3
```

/etc is the directory of the ld.so.cache file.

- Number of environment-application interactions for file

DBStore module counts the number of times the application interacted with a specific environment file.

- Permission of the file

The permission of each interaction file can be captured from the second argument of the line that starts with “open”. Permissions are read and/or write. For instance, in the following line, O_RDONLY shows that this file is just opened to be read.

```
open("/etc/ld.so.cache", O_RDONLY, root) = 3
```

If the file has read and write permission, this argument shows ”O_RDWR” .If the file has only write permission, this argument will be “O_WRONLY”.

- File owner

The file’s owner can be captured from the last argument of the each line that starts with “open” argument. For instance, in the following line, the owner of this file is root.

```
open("/etc/ld.so.cache", O_RDONLY, root) = 3
```

- Start time

Each interaction file starts interacting with the application when the file has been opened. The start time can be captured from the MainOutput file.

- End time

Same as the start time field, the end time can be captured from the MainOutput file.

- File Existence

To make sure that a file starts interacting with the application while it is open, we check the number in front of each line that contains "open" in the beginning. If the value that assigned to the " open' arguments is equal to 3, it means that file has been successfully opened. If the number is –1, it means that the file could not interact with the application. For instance the following line shows that the file interacted with the application and was opened successfully:

```
open("/etc/ld.so.cache", O_RDONLY, root) = 3
```

But the following line shows that the file could not interact with the application:

```
open("/etc/ld.so.preload", O_RDONLY, root) = -1
```

Error Table. Each file can interact with the application at different times. The followings are the fields of the table indicating the status of each interaction.

- File name
- Number of errors

To get the number of times that an error occurred when the application interacted with a file, we scan in the MainOutput file each line that starts with "open". If the number that is assigned after the argument is "-1", it shows that the file could not

interact with the application and that an error had occurred. For instance the following line:

```
open("/etc/ld.so.preload", O_RDONLY, root) = -1
```

shows that ld.so.preload file could not interact with the application. DBStore parses the MainOutput file for each file and increases the counter for each "–1 "it finds and stores this number in this field.

User input Table. This table records user inputs contained in the collector InputFile. User inputs consist of the application execution command and the application directory name where the application is located. The fields of the table are as follow:

- Application Execution Command

The execution command is used to run the application. For instance the "ls" command is the UNIX command to run "ls".

- Application Directory Name

This field contains the directory name where the application is located. For instance in "/bin/ls", /bin is the directory where "ls" is located.

Note that these fields are filled after DBStore is run.

Functional Descriptions

In this part we focus on the functional description of the collector and the dependencies of the functions and associated tasks.

All the modules and files that we described in this chapter are implemented in three main java modules. These modules are Main.java and Basicfilelist.java and DBUtil.

Main.java

Identification : Main.java
Type : Module
Purpose : provides detailed information from system call
Function : Retreive the inputs from the user and stores the data into Inputfile. Then run EAIS module and port the output to the generaloutput file. Following, this function will run the parser module and save the data to the MainOutput file.

Basicfilelist.java

Identification : Basicfilelist.java
Type : Module
Purpose : provides detailed information about interaction files and their properties to the user from the database. It will show all the data in two tables. The first table contains general information about the interaction files. The user can choose any of the files from the table to get detailed properties of the filed that interacted with the application in a new window.

DBUtil.java

Identification : DBUtil.java
Type : Module
Purpose : connects the collector with the Oracle database.

Sequence Diagrams

We can use sequence diagrams to better understand the sequence of actions that are taken in collector to gain information on the environment-application interactions and to store them into the database. With sequence diagram, we can also monitor the life line of each module that was executed. The following figure shows the sequence diagram of the collector and the steps taken to execute this part of Pulad.

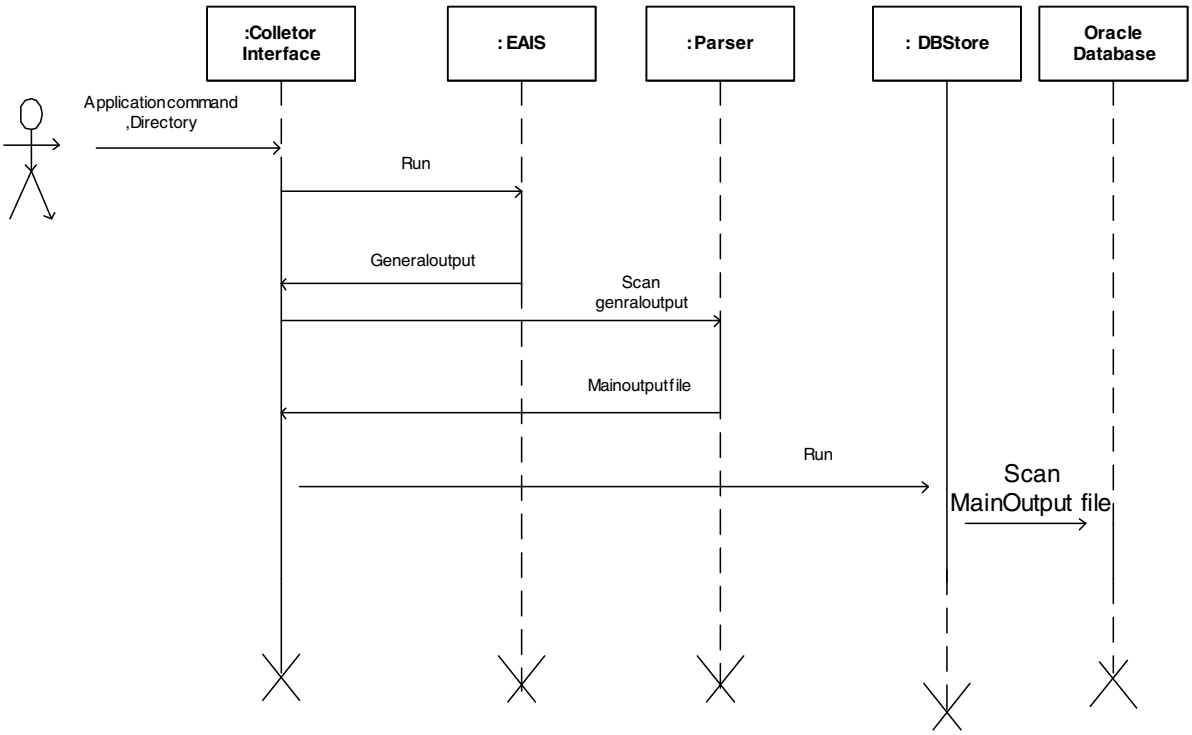

Figure2: Collector Sequence Diagrams

As we can see from the figure, the first module that is executed in the collector is EAIS. This module will be run by the collector and will stop running after saving the output in the GeneralOutput file. The solid line for each module shows the lifeline of that module. Then the collector will run the parser module to remove all unnecessary parts and symbols from the output file that was generated by Java and will store the outcome in the MainOutput file. Afterwards, DBStore module will be run in the collector. This module will port the data to the Oracle database. Each arrow in the figure shows the input and the output of each module.

How to Run Pulad

Pulad is implemented in java using the Eclipse development environment on Linux.

(Eclipse is an open platform for tool integration built by an open community of tool providers.) [Eclipse] We also use Oracle [Oracle] as the database engine to store the data and results. Before running Pulad and executing applications, some preliminary steps are required to connect Pulad with Oracle and Eclipse.

Start the database.

To install Oracle on Linux, a new user account as "oracle" needs to be created. The account can then be used to install Oracle. Once Oracle is installed, it is run oracle using the following commands:

```
$Oracle_Home
sqloracle
```

You then will have to indicate your username and password. Once the sql prompt appears, you just need to type `STARTUP`.

Start the listener

Oracle has a client-server architecture allowing different users to use this database engine simultaneously. Since t users of Oracle are the clients, the client side of Oracle also needs to be run. The following command is needed to run the client side:

```
Oracle_Home
lsnrctl
start
```

Now the oracle client side is ready, we can start running Pulad.

Execute Pulad

First Eclipse needs to be started. Once in the directory of Eclipse, the following commands are run:

```
./eclipse
```

With this command, Eclipse is upload and ready to run Pulad. To execute Pulad, first we should upload the Pulad project. Then by clicking the “Run“command, Pulad will start running.

Conclusions

In this chapter we have described one of the main components our tool for finding application vulnerabilities using fault injection: the collector. The collector records all the environment-application interactions when an application is executed. This information is then used by the fault injector component, which will be described in Chapter 5. In this chapter, we have described the collector at a high level, then have provided design and implementation details. A functional description, sequence diagrams and details on how to execute the collector are also provided.

Chapter 5: Fault Injection for Finding Application Vulnerabilities: Design and Implementation

Introduction

After having described in Chapter 4 the first main component of Pulad called the "collector" that focuses on the collection of environment-application interactions related to files when the application is executed and is called the "collector", we detail in this chapter the second main component that focuses on the injection of faults and is called the "fault injector". More precisely, the architectural view, detailed design and implementation details of the fault injector are described in the following sections.

Overview of the Fault Injector Architecture

The following figure shows the architecture of the fault injector in Pulad.

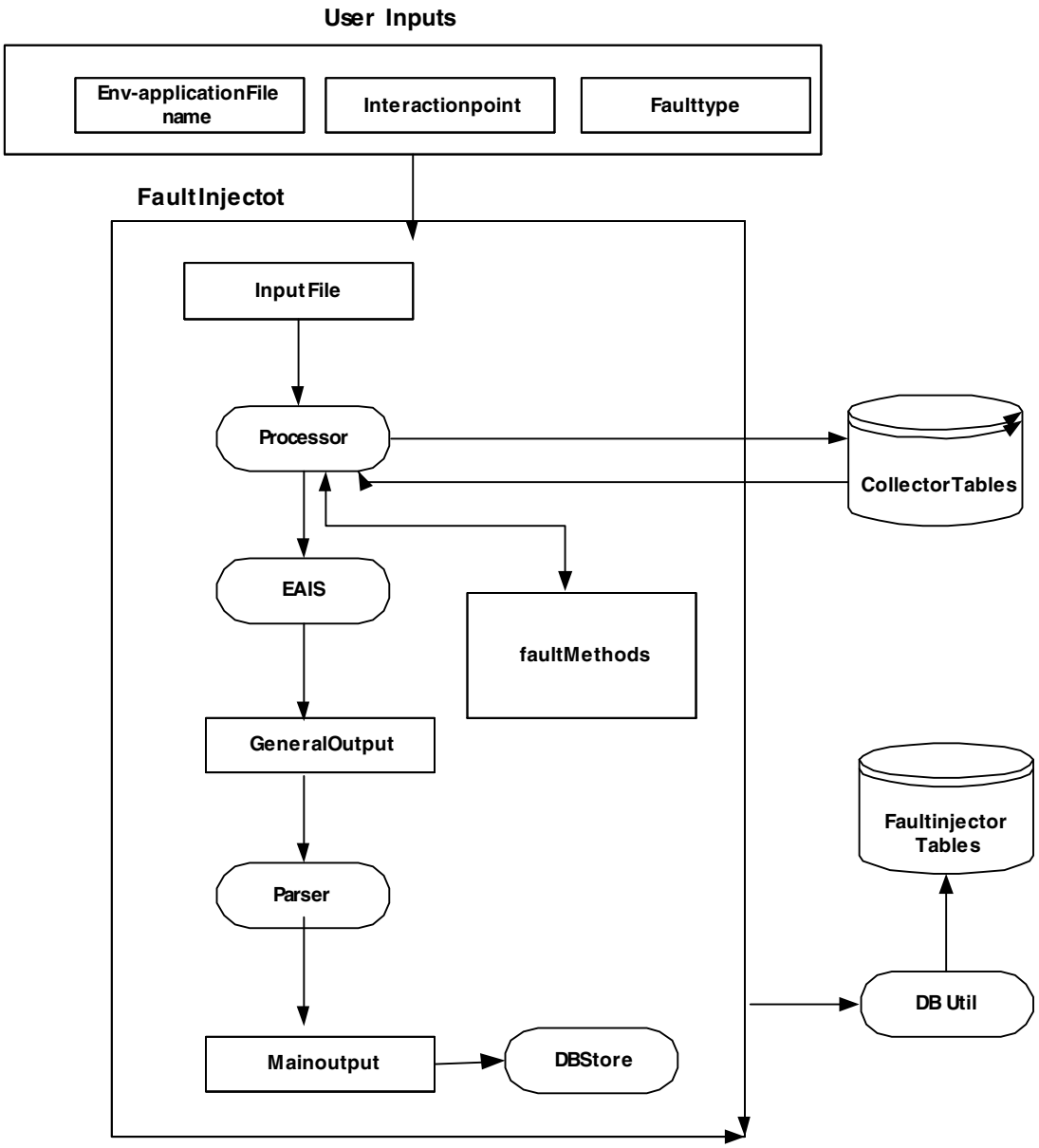

Figure 3: Fault Injector Architecture Overview

In the fault injector, the user provides the inputs either through a file or using a graphical user interface (GUI). The user chooses the interaction file names, faults and the environment-application interaction points in which the faults should be injected either by just clicking the options in the GUI or by creating an input file. All the

inputs will then be stored in an input file.

The processor module then processes the input file. This module searches for each environment-application file with the specific interaction point in the collector's database tables to find all the information regarding that specific environment-application interaction file. The result is returned to the processor. For instance, if the user wants to inject a fault to the File1 at the interaction point 1, the parser searches for the first interaction of File1 with the application and ports the result to the processor. If the processor cannot find such interaction file with the specific interaction point in the database, an error message will appear.

Also the processor searches for the specific fault type that the user chose among the fault methods. The specific fault module associated with a fault type is retrieved and returned to the processor. Note that all fault modules are located in the fault methods file. So the fault module is ready with the interaction file name and the specific information of that interaction point.

Now all the environment-application interaction files and fault types chosen by the user are ready to be injected at the specific interaction time. The EAIS module then runs the application while injecting specific faults just before the selected interactions between the environment and the application. EAIS stores the output in the GenralOutput file. As for the collector, Java has created some extra symbols when executing the application. As described in Chapter 4, the parser module will mainly trace the GeneralOutput file and remove the extra symbols and store the result into the MainOutputfile. Afterwards, the DBUtil module will connect the fault injector to the database. DBStore will then parse the MainOutput file to store the data in the

specific database tables.

Fault Injection techniques for Fault Injector

Each environment-application interaction file might interact with the application more than once. So each file can have many interaction points. To inject faults in specific interaction points, each fault should be injected in the specific environment-application interaction file just before the specified interaction with the application.

For instance, the following table shows the interaction of one application with three different files at different times. The user is interested in injecting one fault (change path) in File1 at interaction point 2, injecting one fault (change directory) in File2 at interaction point 3 and injecting one fault (change name) in File3 at the interaction point 3.

FILE	Interaction 1	Interaction 2	Interaction 3
File 1	0-1	2-10	-
File 2	1-3	6-8	11-15
File 3	0-1	2-9	10-12

Table 8: Example of Input File

The above table shows the interaction file name and the start time and close time of each interaction point. For instance File1 interacts with the application at three different times. The first interaction is between time 0 and time 1. The second interaction is between time 2 till time 10. The two other files also interact with the application at different times. To inject faults at the specific interaction point, each fault should be injected before the interaction starts. Also each fault for each file that interacts with the application several times should be injected after the previous

interaction point. To inject faults in each file at a specific interaction point, we also need the ending time of the previous interaction points. This is the time when we should inject the fault.

As mentioned before, each fault will be injected while the application is running through EAIS. While the application is running, faults will be injected in different files. In the above example the user intends to inject three different faults in three interaction files. Therefore, we should first order these files based on the interaction times to know the order of injection. The files are ordered based on the ending time of the previous interaction point. To do so, we capture the previous interaction points of the files that are involved in fault injection. For instance for File1, since we want to inject a fault at the interaction point 2, we get the ending time of the interaction point 1 of File1 with the application, which is time 1. This is the time from when we should inject the fault to this file. We do the same procedure for the other files, so obtain the following results:

FILES	Time of injections based on previous interaction points
File1	1
File2	8
File3	9

Table 9: Sorting of Files Based on Interaction Times

For File1, time 1 is the ending time of the interaction point 1 (0-1) and time 8 is the ending time of interaction point 2 (6-8). These are the times when different selected faults should be injected. We also have to order these ending times for different files. To do so, we sort them in function of time. Each fault for different interaction point times should be injected to the files while the application is running. For the above

example, the faults should be injected in the following files based on their interaction time in the following order: File1, File3, File2.

Detailed Design of the Fault Injector

Since several modules have already been detailed in Chapter 4, in this section we provide brief description on the detailed design of each file and module of the fault injector.

The Input File

This time, the input file stores the user inputs for the fault injector. User inputs consist of environment-application interaction file names, interaction points and fault types. All the inputs can be retrieved from the user either through a file or a GUI. The first argument of the file is the environment-application file name in which the user wants to inject a fault. These files were stored in the database by the collector. If the file that user inputs cannot not be found in the database, an error will occur. The other user input is the interaction point. As mentioned, each environment-application file can be called more than once by the application. The collector captured the different interaction points as well as the starting and ending time of the interactions for that specific interaction file. The fault should be injected right before the starting time of the interaction. The fault type is another input from the user. The fault type indicates what kind of fault the user wants to inject. Later in this chapter we will describe each fault type in more details.

The following line shows the user inputs from the input file:

libm.so.6 3 changename

The first argument is the name of the environment-application interaction file, which is libm.so.6. The second argument is the interaction point. In this example, 3 means that we want to inject the fault at the third interaction point between the application and libm.so.6. “changename” means that the fault type module (here changename) is changing the name of the file.

The Processor

The processor is the main module of the fault injector. It has four major tasks as follow:

1. Get the application name and the directory name from the collector database.

2. Replace interaction point with the associated start time and ending time.

3. Scan the fault methods file for the specific fault type module for fault injection.

4. Sort all the environment-application files based on the interaction points.

The input of this module for each of these tasks is the Input file. Now we will describe each of these tasks in detail:

Get the application name and the directory name from the collector database. The first task consists of retrieving the application name and the directory name that was stored in the Collector’s database. The processor gets the inputs from the input file. The first argument in the input file is the environment-application interaction file name. These interaction files were captured during the execution of the collector and stored in their specific tables in the database. This module queries the tables to find specific environment-application interaction files and their attributes at a specific

time. If this environment-application interaction file in the specific interaction point exists, then the fault injector ports all the information regarding this specific file back to the processor. If the processor could not find the specific environment-application file in the specific interaction point, then the fault injector shows an error message.

Replace interaction point with the associated start time and ending time. The other task of the processor is to replace the interaction point number with the start interaction time and the end time of the specific environment-application file. These two times give the interval time of the specific file interaction with the application. So the processor, after retrieving all the information about the environment-application interaction file at a specific interaction point, starts replacing the interaction point with the start time and the end time of the file. The processor operates this task, because each fault injection should occur just before the starting time of the interaction. For instance if the interaction point is 2 (which means the second interaction of the application and the specified file), the second start time and end time of the file will be retrieved from database. Also the fault injection, while running the application again, should happen just before the second starting time of the interaction with the specific file.

Scan the fault methods file for the specific fault type module for fault injection. The processor gets the fault type from the input file. Then it scans the specific fault module from the fault methods file. The fault methods file contains all the fault modules that inject faults to the interaction files. Later in this section we will describe the fault methods file more in details.

Sort all the environment-application files based on the interaction points. Note

that the fault injections and the EAIS module run simultaneously. So in order to inject faults before the staring time of each interaction, we need an algorithm to sort the interaction files. The algorithm we developed is based on the interaction time. This algorithm orders the faults based on the start time and the end time of the interaction. This algorithm sorts the list of files in which a fault will be injected in an increasing order of the start time and then increasing order of the end time. The only special case is when a fault should be injected at time zero. This case happens when the application starts interacting with the specific interaction file by the time it starts executing. In this case the fault should be injected before the execution of the application.

The following summarizes the algorithm developed.

1) Get the interaction file name, interaction point and fault module from Inputfile.

2) Replace interaction point (IP) from Inputfile with time intervals

(start_time, end_time)

3) Sort interaction files by increasing start_time

4) Sort interaction files by increasing end_time

5) Special case: If there is any IP, which is equal to one, inject the faults now

After ordering the interaction files based on the time, the fault injector is ready to inject faults in the application while running the EAIS module.

The EAIS, Parser, and DBStore Modules

While injecting faults into the application, the EAIS module will be executed to collect the environment-application interactions related to interaction files. The output

of the EAIS is stored in GeneralOutput file. As mentioned in Chapter 4, this file contains some symbols that were created with java. The parser will then parse this file and store the output in MainOutputfile. (The details of the parser were provided in Chapter4). Then the DBUtil module will connect the application to the database. And DBStore will port each of the interaction files to the fault injector database.

The fault injector tables in the database are the same as the collector tables with the same fields but different names. The difference between these tables is that the collector tables contain the data before fault injections and the fault injector tables contain the data after fault injections.

The Fault Methods File

In order to inject faults in the environment-application interaction files, we need a fault module to modify specific environment-application files before the start time of the interaction. Each module injects one specific fault type. The fault methods file contains all these modules. The modules are as follows:

1) Change file size.

   Insert characters such as “/”,”>”,”<” ,”|”,”&” or any other symbols. Delete some characters to increase the size of the file.

2) Change file name.

   Insert characters such as “/”,”>”,”<” ,”|”,”&” or any other symbols. Delete the name of the file

3) Change path.

   Create the same file with the same content in another directory. Delete the file

in the directory.

4) Change ownership.

   Change the owner of the file.

5) Change permission.

   Change the permission of the file from read to write, or read and write. Flip the permission bit.

6) Change content.

   Insert characters such as “/”,”>”,”<” ,”|”,”&” or any other symbol. Delete some characters to decrease the size of the file

Functional Description

In this part we focus on the functional description of the fault injector and the dependencies of the functions and associated tasks.

All the modules and files that we described in this chapter are implemented in three main java modules. These modules are Fault_Main.java and Fault_Filelist.java, Fault Module, and DBUtil.

Fault_Main.java

Identification	: new_Main.java
Type	: Module
Purpose	: Inject faults to the specific files and provides detailed information from system call after fault injection.
Function	: Runs the processor to retrieve the application execution command and the application directory from the database and gets fault type, file name and the interaction point from Inputfile. EAIS runs the application while injecting faults. The output is saved in output.txt. The parser scans each line and finds the interaction’s attributes and ports them to the database through DBStore module.

Fault_Filelist.java

Identification : Fault_Filelist.java
Type : Module
Purpose : Provides detailed information about interaction files and properties after fault injection. The information from the database is presented in two tables. The first table contains general information about the interaction file, number of interaction points and fault type. The second table gives details on each interaction points, start time and end time.

Fault Module

Identification : FaultModule
Type : Module
Purpose : Contains different modules of fault models
Function : Injects faults to the file based on user demand.

DBUtil.java

Identification : DBUtil.java
Type : Module
Purpose : Connects Pulad to the Oracle database.

5.6 Sequence Diagram

We can use sequence diagrams to better understand the sequence of actions that are taken by the fault injector to inject faults in the environment-application interaction files and store the result in the database. With a sequence diagram, we can also monitor the lifeline of each module that was executed. The following figure shows the sequence diagram of the collector.

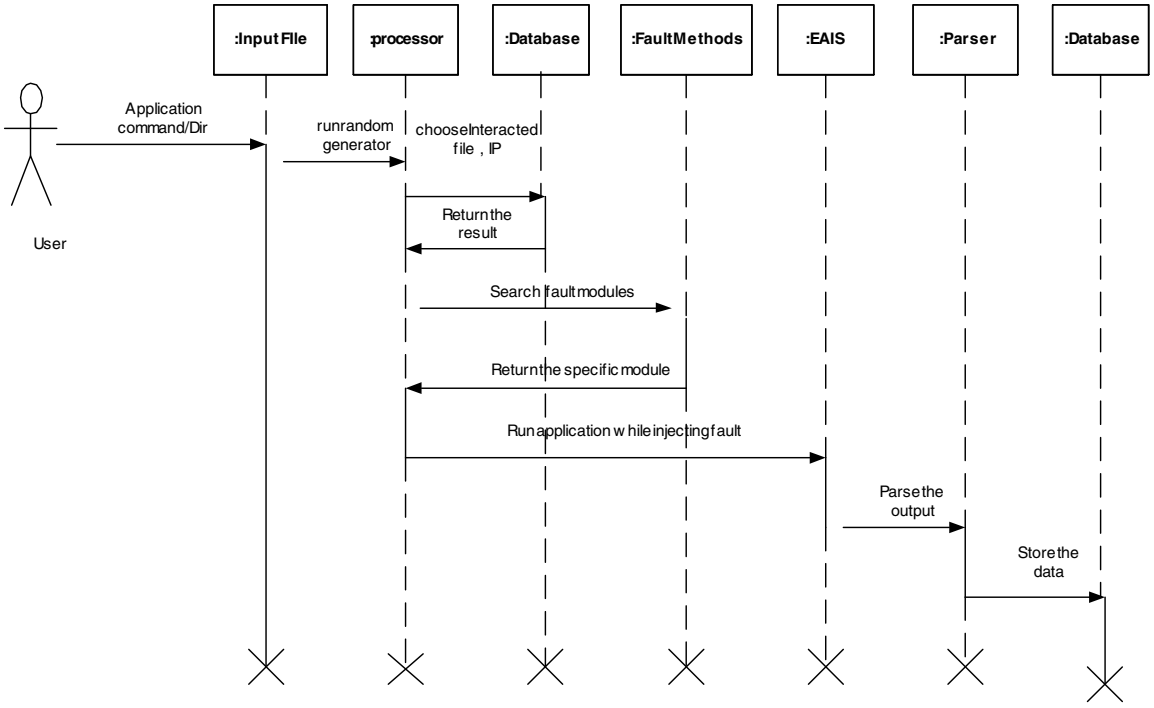

Figure 4: Fault Injector Sequence Diagrams

As we can see, first the user enters the inputs to inject faults. These data are stored in an input file. Then the processor scans the collector to retrieve the interaction files that the user entered as an input for fault injection for the specific interaction points. Thereafter, the processor searches for the specific fault modules from the fault methods file to inject the faults. All these data return to the processor and all files are reordered based on the fault injection algorithm that Pulad is using. Then the application runs through EAIS while faults are injected at the specific time. The output is stored in the GeneralOutput file. The parser will parse all the data and remove all unnecessary information and store the result in the MainOutput file. At the end the DBStore module scans the MainOutput file and ports specific data into the database. The solid line for each module shows the lifeline of each module. Also each

arrow indicates the input and the output of each module.

Conclusions

In this chapter, we detailed the second main component that focuses on the injection of faults and is called the “fault injector”. More precisely, the architectural view, detailed design and implementation details of the fault injector were described in this chapter. Now that we have detailed the design and implementation of Pulad, we describe how we validated Pulad in the next chapter.

Chapter 6: Verification of Pulad

Introduction

We introduced Pulad in Chapter 4 and 5. In this chapter we describe how we validated Pulad. The following concepts are taken from [Lap98]. The validation process consists of removing faults or bugs and predicting the behavior of the system relative to the occurrence of faults and their activation. This chapter only focuses on the removal of faults. Fault removal consists of three steps: verification, diagnosis, and correction. Verification is the process of checking whether the system adheres to properties, termed the verification conditions [Che81]. If not, the two other steps must be undertaken: diagnosis of fault(s) preventing the verification conditions to be met and then, performing the necessary corrections. Following correction, the process must be repeated to ensure that fault removal has not entailed undesirable consequences. The verifications thus performed are termed as non-regression.

We applied in this chapter only software testing, which is the most popular dynamic verification method. Software testing is a process used to identify the correctness, completeness and quality of developed computer software. Actually, testing can never establish the correctness of computer software, as this can only be done by formal verification (and only when there is no mistake in the formal verification process). [SoftwareTesting] The methods for the determination of the test patterns can be divided into several classes according to two viewpoints: criteria for selecting the test

inputs, and generation of the test inputs.

The techniques for selecting the test inputs may in turn be classified according to three viewpoints:[lap98]

- The purpose of the testing: checking whether the system satisfies its specification is conformance testing, whereas testing aimed at revealing fault is fault-finding testing;
- The system model: depending on whether the system model relates to the function or the structure of the system, leads respectively to functional testing and structural testing;
- Fault model: the existence of a fault model leads to fault-based testing, aimed at revealing specific classes of faults.

According to the approaches considered, test input generation may either be deterministic or probabilistic:

- In deterministic testing, test sets are determined by a selective choice according to the criterion retained,
- In the random testing, test sets are selected according to a probabilistic distribution of the input field, the distribution and number of data inputs being determined in accordance with the criterion retained.

Selected Testing Techniques

For Pulad, we applied boundary testing, functional testing and stress testing.

Pulad gets different inputs from the user, either in the collector or in the fault injector. Wrong inputs might be considered as a fault and lead to the failure of Pulad. So

boundary testing will compare Pulad's inputs, which were entered by the user, with the expected results that we had obtained through the collector.

We also applied functional testing, a black-box testing technique, to ensure that each function runs and performs its task as expected.

The other testing method we chose is stress testing because it is applicable to programs that operate interactively. We chose this technique for Pulad to check the behavior of the application based on the overload of user inputs.

Boundary Testing

" The systematic testing of error handling is called boundary testing Boundary testing refers to the testing of forms and data inputs, starting from known good values, and progressing through reasonable but invalid inputs all the way to known extreme and invalid values [Bei90]. The logic for boundary testing forms is straightforward. We start with known good and valid values because if the system fails on that, it's not ready for testing. Next we move through expected bad values because if those fail, the system isn't ready for testing. Then we try reasonable and predictable faults because users are likely to make such faults. Then start hammering with extreme faults and inputs in order to catch problems that might affect the tool's functioning " [Bei90].

Functional testing

The other way of verifying the Pulad application is using another testing technique called functional testing. Functional testing is a process of attempting to find

discrepancies between the program and the external specification. External specification is a precise description of the program's behavior from the point of view of the end user [Mye04]. In other words, it is a technique to check and verify each module and function in the source code and to compare the result with what is expected. To perform a functional test, the specification is analyzed to derive a set of test cases. We supply the test cases as an input data; the output data (if applicable) or an error code (if the input data is not valid) is expected.

Stress testing

Stress testing is a form of testing which is used to determine the stability of a given system. It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results. Stress testing subjects the program to heavy loads or stresses. A heavy stress is an important volume of data, or activity, encountered over a short span of time. To perform the stress testing, the specifications are analyzed and the test cases are created based on the specifications. In each test case, the heavy load (often to a breaking point) of data will be run as an input data [Mye04].

Examples of Applications

We chose different applications and run them through Pulad to observe different results. First, we chose a simple UNIX command, called ls. We chose this application because it is small and runs in any directory in UNIX. The ls application interacts with different files like ld.so.preload , libtermcap.so.2, ld.so.cache, libtermcap.so.2,

libc.so.6, ld.so.cache and libm.so.6 that would be captured by Pulad.

To have a better control on Pulad and to validate it, we implemented two applications. In these two applications we know exactly what files these applications are interacting with. So we can track the results obtained by Pulad with the base information about the application implementation. The first application interacts with 5-6 environment–application interaction files. Each of the files interacts twice with the application. We chose one file from each file type. For instance we chose one library file, one environment file, and one temporary file. So the first application starts calling each file, gets the ownership, directory and the permission of each file and ports the result to the terminal. So the user can monitor the result after modifying each file on the terminal.

The second application that we implemented interacts with more than ten environment-application interaction files (3 files from each file type) and each file was called more than two times by the application. In this application, each file that was called interacts 100 times with the application. Note that each interaction occurs at different times, so we have more than one interaction in different time intervals.

We will call one of the applications a “small application” because consists of about one hundred lines of code and interacts with a few files, and the other one a “large application” because it consists of more than a hundred lines of code and interacts with more files.

Testing Results

We used three testing methods on the application examples (ls, small application and

complex application). We now present the results we have obtained.

Boundary Testing

We used this technique to test the collector and the fault injector of Pulad with the small application and the ls command. In the first step of boundary testing, we used the correct inputs from the user, which was the running command of the application and the directory. For instance for the ls command, we used "/bin ls" as an input and we run Pulad. Then we modified this input. For example we changed the length of the directory, or we inserted a directory name that did not exist, to verify the error message at the right time. Then entered inputs to check the boundary, for instance long directory path or long directory names for the application.

When focusing on the fault injector, first we ran the application with the correct inputs, which consists of the interaction file name, interaction point and fault type. Then we entered a wrong type of data, for instance a wrong interaction file name, or an interaction point that did not exist, or a fault type that did not exist in the fault methods file. So an error message for these cases is expected. Then we checked the boundaries by entering long or short input data.

With this technique all the test cases based on the description of the inputs were examined and compared with the definition of each module and output. The following table shows the 23 test cases we used for the "ls" command with boundary testing. We used the same test cases for the small and large applications.

LSR	Test Cases	Expected Result	Application Result	Fail/ Pass	Bug Fixed

1	Entering number in User input , in application execution command in collector	Error Message" No Such choice" displays	As expected	Pass	-
2	Entering numbers in application directory in collector	Error Message" No Such choice" displays	As expected	Pass	-
3	Entering long directory as an input in collector	Error Message" long path" displays	As expected	Pass	-
4	Entering long application execution command in the collector	Error Message" long path" displays	As expected	Pass	-
5	Entering short directory path as in put in the collector	Error Message" Short Path" displays	As expected	Pass	-
6	Entering short execution command (less than what is expected) as in put in the collector	Error Message" Short Path" displays	Shows error	Fail	-

7	Entering numbers in Env-application file name in Fault Injector	Error Message" No Such choice" displays	As expected	Pass	-
8	Entering characters in Interaction Point (IP)as input in Fault Injector	Error Message" No Such choice" displays	As Expected	Pass	-
9	Entering characters in IP in the collector	Displays error	Accept the input	Fail	Y
10	Entering IP =0 in fault injector	Display Error	Accept the input	Fail	Y
11	Entering long number more than 100 in fault injector	Display Error	Accept input	Fail	Y
12	Entering number as a fault type in Fault injector	Display Error	As expected	Pass	-
13	Entering long name as a fault type in fault injector	Error message" Invalid fault type"	As expected	Pass	-
14	Entering Wrong Fault type in fault injector	Error message "Invalid fault type,"	As expected	Pass	-
15	Entering blank in IP in fault injector	Error displays	As expected	Pass	-

16	Entering blank in Env-application name in fault injector	Error displays	As expected	Pass	-
17	Entering blank in Fault type in fault injector	Error message	As expected	Pass	-
18	Entering blank in application execution command in collector	Error message	As expected	Pass	-
19	Entering bank in application directory in collector	Error message	As expected	Pass	-
20	Entering long Env-application file name	Error displays	As expected	Pass	-
21	Entering two application execution command as input in collector	Error displays	As expected	Pass	-
22	Entering no information as an input for fault injector	Error displays	As Expected	Pass	-
23	Entering numbers in fault type in fault injector	Error displays	As expected	Pass	-

Functional Testing

We used functional testing to make sure that each module in Pulad is working and

that the result is as expected. So each of the modules that were described in Chapter 4 and 5 was tested and run. The result of each module was compared with the purpose and goal of each module. So for the collector and the fault injector, we tested each of the modules that we described based on the input and the output of each file and compared the result with what we expected from the definitions and the structure in Chapter 4 and 5.

The following table shows the 17 test cases that we created to compare each module's result with the expected result. We run these test cases three times with the three applications we have.

LSR	Test Cases	Expected Result	Application Result	Fail/ Pass	Bug Fixed
1	Entering application directory path and execution command to check if it will be stored in an input file	Get the input and port it into the input file	As expected	Pass	-

2	Providing application directory path and the execution command to check EAIS module	Get the input and port the output into the General out put file	As expected	Pass	-
3	Providing EAIS result in Genral- Output file to test Parser module	Get the input, scan and delete extra symbols	As expected	Pass	-
4	Providing Main out put file to check DBStore module	Get the file, scan the output file and get the information for each specific field	As expected	Pass	-
5	DBUtill module	Connect the parser to Oracle database	As expected	Pass	-

6	Providing fault type , file name and IP to check if the input file stores them	Get the input and stores it into Input file	As expected	Pass	-
7	Provide all the info to check the processor	Get the input and starts scanning through collector database	As expected	Pass	-
8	Provide file name to the processor through input file	Get the file name and search for the file name through collector database	As expected	Pass	-

9	Provide Interaction point to the processor through input file	Get the interaction point and search for the specific interaction point for the specific file name through collector database	As expected	Pass	-
10	Provide Fault type to the processor through input file	Get the fault type and scans it through fault module	As expected	Pass	-
11	Given a fault type to the fault module to check the fault type module	Get the fault type and scans its through its functions and port the specific function to the fault type module	As expected	Pass	-

12	Given all the inputs from the Processor to EAIS to check EAIS module	Get the inputs from the processor and the collector database and runs the application	As expected	Pass	-
13	Provide all the data from EAIS in General Output file to check Parser module	Get general out put file , scans it , delete extra symbols and port the result in the Main output file	As expected	Pass	-
14	Provide all the inputs for DBStore ready to check DBStore function	Get the inputs from Main out put file and scan it and port the data to the database	As expected	Pass	-
15	Heck DBUtill module	Connect the Fault injector and oracle database	As expected	Pass	-

16	Providing inputs in to the input file to port the database to the database collector	Get the inputs to the input file and port the data to the collector database	As expected	Pass	-
17	Providing application directory and the execution command to check EAIS module in fault injector	Get the application command and the directory from the collector database	As expected	Pass	-

Stress Testing

We ran the collector with the "ls" command application and used the stress testing.We created the test cases for stress testing with the high volume of inputs to check the behavior of the collector and the fault injector.

We ran the "ls" command though the Pulad with the following testcases and compared the result with the expected result.

LSR	Test Cases	Expected Result	Application Result	Fail/ Pass	Bug Fixed
1	Entering two application name and directory in the collector	Error message " only one application to be run at the time"	As expected	Pass	-
2	Entering all fault type to all files in all interaction points	Inject faults before IP was reached	As expected	Pass	-
3	Entering all faults to one IP of a file	Inject faults before IP was reached	Error	Fail	Y
4	Entering all faults to all file in all interaction points	Inject faults before IP was reached	Error	Fail	-

Conclusions

After having described the development of Pulad in Chapter 4 and 5, we detailed in

this chapter some of the tests that were conducted on Pulad to verify it.

Chapter 7: Conclusion and Future Work

This thesis introduces a fault injector, called “Pulad”, specifically developed for finding application vulnerabilities. Most previous approaches for finding application vulnerabilities involved static verification methods. With these methods, the source code is not executed. Since vulnerabilities can only be revealed when they are exploited, the use of a dynamic verification method, executing the source code, seems needed. We have shown in Chapter 2 of this thesis that static analysis tools would not only identify as vulnerabilities bugs that cannot be exploited (leading to false alarms) but also miss vulnerabilities that could have been found had a dynamic verification method been used (leading to false positives). Therefore, the use of dynamic verification methods that would complement static verification ones is a natural research thread. The main two dynamic verification areas are software testing and fault injection. This thesis focuses on fault injection building upon some preliminary research conducted by [Du00].

The approach introduced by [Du00] focuses on environment-application interactions. These interactions can be used by an attacker to launch attacks on the application. Therefore, a secure application needs to be able to tolerate perturbations of the environment. If we now consider environment perturbations as faults, we then consider a secure system as a fault-tolerant system able to tolerate faults in the environment. We revised the classification given by [Du00] of environment direct

and indirect faults for the network, processes and files. The approach described in [Du00] requires extensive manual analysis of the source code. To provide automation and to remove the assumption on the need of source code, we focused on the sole files and removed the distinction between direct and indirect environment faults.

In Chapter 4, we described Pulad focusing one of the main components of Pulad called the "Collector". The goal of the collector is to record all the environment-application interactions when the application is running. These interactions focusing on the environment files are then analyzed and the following fields are uploaded into a database including the file name, file extension, file size, file directory, number of times file used, file permission (includes symbolic link and ownership) and number of times an error occurred. The chapter first describes the design goals of Pulad, then an overview of the collector and finally some implementation details.

The next chapter then focuses on the fault injector, the second main component of Pulad. The fault injector allows to inject faults either using a graphical user interface (GUI) or directly through a text file. The faults in the files include the file name, the directory name, the execution path, the library path, the file existence, the file ownership, the file permission, etc. For each of the faults, the specific type of fault needs to be indicated. Moreover, the interaction points where the faults should be injected are also provided by the user.

Chapter 6 then describes the validation process we have used to validate Pulad. The

validation of the tool mainly relied on software testing. More precisely, functional testing, boundary testing and stress testing were applied. Running the “ls” command, a simple application interacting with some files of the environment and a larger application interacting with a significant number of environment files were developed. These applications were then used to apply the three testing techniques. Details on the test cases and testing results are provided in Chapter 6.

The next step in finding application vulnerabilities using fault injection is to apply Pulad on a real application to find vulnerabilities not identified yet. Issues related to this research include the choice of the fault types and the number of faults to inject, the interaction where to inject them. However, these issues are already present when dealing with fault injection in fault-tolerant systems. An issue specific to fault injection for proving that a system is secure, is the definition of a security violation when limited knowledge on the system is provided. This is a challenging question that needs to be answered in order to be able to provide some automation and remove the need to having the application source code.

Appendix A: Code of the tracking system written in C:

```
/*Tracker.c*/
#include <stdio.h>
#include <stdlib.h>

int stackoverflow(char * strPara)
{
    char strLocalVar [6];

    strcpy(strLocalVar, strPara);

    return 0;
}

int heapoverflow(char * strPara)
{
    char * strHeapVar;

    strHeapVar = (char *)malloc(sizeof(char)*6);

    if(!strHeapVar){
      printf("The heap runs out of space!\n");
      exit(-1);
    }

    strcpy(strHeapVar, strPara);

    return 0;
}
```

Bibliography

[Asl96] T. Aslam, I. Krsul, and E. H. Spafford, "Use of a taxonomy of security faults," in *Proc. 19th National Information Systems Security Conference*, (Baltimore, Maryland, USA), pp 551-560, October 1996.

[Bei90] B. Beizer, "Software Testing Techniques," Second Edition, New York, Van Nostrand Reinhold, 1990.

[Che81] M.H. Cheheyl, M. Gasser, G.A. Huff, J.K. Miller, "Verifying Security", *Computing Surveys*, vol. 13, no. 3, pp. 279-339, 1981.

[Cla95] J. Clark and D. Pradhan, "Fault injection: A method for validating computer-system dependability," in *IEEE Computer*, pp. 47-56, June 1995.

[Cow99] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," in *Proc. DARPA Information Survivability Conference and Expo (DISCEX)* (Co-sponsored by the IEEE Computer Society), vol. 2, pp. 271 –283, 1999.

[Dan99] T. E. Daniels, B. A. Kuperman, and E. H. Spafford, "Penetration Analysis of a XEROX Docucenter DC 230ST: Assessing the Security of a Multi-purpose Office Machine," in *Proc. 23rd National Information Systems Security Conference*, (Baltimore, MD, USA), October 2000.

[Dow01] D. D. Downs and R. Haddad, "Penetration Testing – The Gold Standard for Security Rating and Ranking," in *Proc. First Workshop on Information Security System Rating and Ranking*, (Williamsburg, Virginia, USA), May 2001.

[Du00] W. Du and A.P. Mathur, "Testing for Software Vulnerability Using Environment Pertubation," in *Proc. International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, (New York City, New York, USA), pp. 603-612, June 2000.

[Du98] W. Du and A.P. Mathur, "Vulnerability Testing of Software System Using Fault Injection," COAST, Purdue University, 1998.

[Eclipse] http://www.eclipse.org

[Flawfinder] http://www.dwheeler.com/flawfinder/

[Gar96] S. Garfinkel and G. Spafford, "Praxtical UNIX & Internet Security," O'Reilly & Associates, Inc., 1996.

[Gho98] A. K. Ghosh, T. O'Connor, and G. McGraw, "An automated approach for identifying potential vulnerabilities in software," in *Proc. 1998 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, pp. 104-14, 1998.

[Kak00] R. Kaksonen, M. Laakso, and A. Takanen, "Vulnerability Analysis of Software through Syntax Testing," Technical Research Center of Finland, http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/

[Krs98] I. Krsul, "Coast vulnerability database reference guide – draft version, Technical report, Purdue University, Department of Computer Sciences, 1998.

[Lap98] Dependability Handbook: Preliminary Version, Ed. J.-C. Laprie, LAAS Report no 98-346, 1998.

[MAFTIA00] C. Cachin et al., "Reference Model and Use Cases," in *MAFTIA deliverable D1*, 2000.

[MAFTIA01] “Conceptual Model and Architecture,” MAFTIA deliverable D2, D. Powell and R. Stroud Ed., November 2001.

[McG98] G. McGraw, “Testing for Security During Development: Why We Should Scrap Penetrate-and-Patch”, in *IEEE AES Systems Magazine*, April 1998.

[mingetty] http://ourworld.compuserve.com/homepages/KanjiFlash/mingetty.c

[Mye04] G. J. Myers, C. Sandler, T. Badgett, T. M. Thomas, “The Art of Software Testing,” John Wiley & Sons, 2 edition, 2004.

[Oracle] http://www.oracle.com

[Pan05] S. Panjwani, S. Tan, K. Jarrin, and M. Cukier, “An Experimental Evaluation to Determine if Port Scans are Precursors to an Attack,” in *Proc. International Conference on Dependable Systems and Networks (DSN-2005)*, Yokohama, Japan, June 28-July 1, 2005, to appear.

[PROTOS] http://www.ee.oulu.fi/research/ouspg/protos/index.html

[RATS] http://www.securesoftware.com/rats.php

[SoftwareTesting] http://encyclopedia.laborlawtalk.com/Software_testing

[strace] http://www.liacs.nl/~wichert/strace/

[Vie02a] J. Viega and G. McGraw, “Building Secure Software,” Addison-Wesley, 2002.

[Vie02b] J. Viega, J.T. Bloch, T. Kohno, and G. McGraw, “ITS4: A Static Vulnerability Scanner for C and C++ Code,” in *Proc. Annual Computer Security Applications Conference*, Las Vegas, NE, USA, December 9-13, 2002, to appear.