Fuzz Testing in Stack Based Buffer Overflow

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By

Manisha Bhardwaj
(Roll No. 801531008)

Under the supervision of:

Dr. Seema Bawa
Professor
Computer Science and Engineering Department

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA-147004

July 2017

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Fuzz Testing in Stack Based Buffer Overflow*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Seema Bawa* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Manisha Bhardwaj)

(801531008)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

12/07/2017

(Dr. Seema Bawa)

Professor, CSED

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"***Fuzz Testing in Stack Based Buffer Overflow***"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Seema Bawa* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Manisha Bhardwaj)
(801531008)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Seema Bawa)
Professor, CSED

Abstract

Due to rapid deployment of information technology, the threats on information assets are getting more serious. These threats are originated from software vulnerabilities. The vulnerabilities bring about attacks. If attacks are launched before the public exposure of the targeted vulnerability, they are called zero-day attacks. These attacks damage system and economy seriously. One such attack is buffer overflow attack which are threat to the software system and application for decades. Since buffer overflow vulnerabilities are present in software so attackers can exploit thus obtains unauthorized access to system. As these unauthorized accesses are becoming more prevalent, so there is need for software testing to avoid zero-day attacks. One such testing is fuzz testing, locates vulnerabilities in software and find deeper bugs.

The Stack based-American Fuzzy Lop(SAFAL) model has been proposed. This model works for software to exploit vulnerabilities. The model begins the process of fuzzing by applying various modifications to the input file. The binaries are compiled using the AFL wrappers. Input test case file is provided to the model to execute the test cases. The target program resulted in various crashes and hangs, that discovered stack buffer overflow vulnerabilities. The list of crashes, hangs, queues are found in output directory. The model displays real-time statistics of the fuzzing process. The SAFAL model improves the quality of software as the hidden bugs are found. The effectiveness and efficiency of SAFAL model is hence established.

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life. It is a great privilege to express my gratitude and admiration towards my respected supervisor **Dr. Seema Bawa** Professor Computer Science & Engineering Department. She has been an esteemed guide and great support behind achieving this task. This work would not have been possible without the encouragement and able guidance of her. I also thank my supervisor for her time, patience, discussions and valuable comments. Her enthusiasm and optimism made this experience both rewarding and enjoyable. I am truly grateful to her for extending her total co-operation and understanding whenever I needed help and guidance from her. I am also heartily thankful to **Dr. Maninder Singh**, Associate Professor and Head, Computer Science & Engineering Department and **Dr. Rupali Bhardwaj**, PG coordinator, for motivation and providing uncanny guidance and support throughout the preparation of the thesis report.

I will be failing in my duty if I do not express my gratitude to **Dr. S. S. Bhatia**, Senior Professor and Dean of Academic Affairs, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to staff members of Computer Science and Engineering Department for their help, cooperation, love and affection, which made my stay at Thapar University memorable. Last but not least, I would like to thank my family for their wonderful love and encouragement, without their blessings none of this would have been possible.

Manisha Bhardwaj

(801531008)

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Manually doing exploitation is one of the difficult and time-consuming process as it requires low-level knowledge of computer systems and also analyzes the control and data flow of program. If the program is complex, analysis work will become difficult.

In software testing field, many techniques aim to find bugs in a program and generate test case to trigger those bugs. However, those test cases are meaningless and manual exploit generation is a difficult and time-consuming process because it not only requires low-level knowledge of computer systems, such as operating system internals and assembly language, but also analyzes the control and data flow of program execution by hand. If the program under test is large or uses complex algorithms, the analysis work will be extremely difficult.

In software testing field, there are unknown vulnerabilities in program and by the generation of test cases, these vulnerabilities have been found that led to the less failure of the system in undesirable circumstances. For this reason, the work done here shows the exploitation of stack based buffer overflow in software and fuzz testing to find the vulnerabilities which are threat to the system.

1.1 Software Testing

Software Testing evaluates the usability or capability of a program. It is a process where maximum errors are to be found, as execution of program proceeds which aim at getting zero defect software. Software testing is an important phase in a software for accessing the quality. Software testing involves verification, validation and error detection. These techniques are involved so that the problems in the software gets fixed.

1.1.1 Verification

Verification in software testing describes the behavior of the system as per the specified requirements. It checks and tests for conformance and consistency of software are as per defined requirements. In this generally the question, if we are building the product right.

1.1.2 Validation

System correctness is checked according to the what user has specified and what the user wants. In this generally the question is that the system being build is a right system or not.

1.1.3 Debugging

Debugging is an integral part of software development lifecycle. It involves the location and correction of errors in a computer program.

1.1.4 Error Detection

Error detection determines the number of error that have been detected. To determine how the system reacts on wrong and right inputs, a number of test cases are developed and a number of testing is done**.**

1.2 Software Testing Methodologies

Software testing methodologies involves integration of tests case design methods with planned steps that results in successful testing of a software. The testing strategy is generally designed by project manager, testing specialist and software engineers. The testing strategies are:

i. Black-box testing - In black box testing knowledge regarding the internal design or code is not necessary. Testing is done on the basis of requirements and functionality.

i. White-box testing - White box testing is done when internal logic of code is known. Code coverage, path, branch and conditions are tested.

ii. Unit testing - Testing is done at the 'micro' level. Testing of function or modules is generally done by programmers and not testers because program's internal design and code knowledge is required.

iii. Incremental integration testing - Every time some new functionalities are added because there is requirement of continuously testing the application. Tests the application independently so that functionality of each module is tested before the program is tested completely. This testing is carried out by both programmers or testers.

iv. Integration testing - In this type of testing, testing of modules is done together. This determines the functionality of modules working together. This type of testing preferable for distributed system and client/server.

v. Functional testing - Functional requirements of an application is tested which is generally done by the testers.

vi. System testing - Overall requirement specifications are tested and as a whole system is tested i.e., all parts or modules of the system are tested.

vii. End-to-end testing - The tests are performed at the "macro" level. End-to-end testing involves a complete environment for an application similar to a situation that replicates the real world, such as database interaction, hardware, system, or network communication.

viii. Regression testing - Retesting of software even after fixing or modification of software or environment.

ix. Acceptance testing - Acceptance testing is the final testing stage where the specification of software is tested and is performed by end-user or customer.

x. Load testing - Load tests are performed when applications are under heavy loads, for example when testing a website that has a range of loads thus determine the response time.

xi. Stress testing - Stress is performed when the system is under heavy loads, repetition of some actions or inputs, when inputs have large numerical values, complex queries in database system, shortage of resources.

xii. Recovery testing - If there are crashes, hardware failure this testing describes the system recovery

xiii. Security testing - If there are any unauthorized access such as internal or external access so the security testing describes how the system protect against such access.

xiv. Compatibility testing - This type of testing describes how the system will perform in hardware, software, operating system or network environment.

xv. Exploratory testing - This type of testing generally does not have test plan or test cases. It is a creative, informal software test where testers learns as the software is tested.

xvi. Ad-hoc testing - Ad-hoc testing is kind of exploratory testing but the testers before testing have understanding of software.

xvii. User acceptance testing - This testing is done for an end-user or customer for the satisfaction of software.

xviii. Alpha testing - Alpha testing is done by the end-users, tests the application when development is near to completion so minor changes may be done as a result of testing.

xix. Beta testing - Beta testing is done by end users, testing is done when the development is completed. Bugs and problems are to be found before the final release

xx. Mutation testing - This testing determines if in a given set of test data or test cases, some changes in the code are introduced and then retesting the data or cases to determine if the bugs are detected .

xxi. Gray Box Testing - Gray-Box testing combines both black and white box testing. If there is improper usage or structure of application, gray box testing search for the defects. A black-box tester is not aware of internal structure and white-box tester has access to the internal structure of the application. A

gray-box tester partially knows the internal structure, which includes access to the documentation of internal data structures as well as the algorithms used.

1.3 Need of Fuzz Testing

Fuzz testing is a technique which tests an application in a way that breaks the program. So in traditional software testing, a software developer writes an application and is supposed to perform some functions. For example, a calculator application. Testing of computer application, the requirement is to actually do the math correctly. If the application is working properly, then that is usually enough to pass the traditional tests.

The difference with fuzzing technique is that the fuzz tester provides invalid input to an application and cause the application to misbehave. Thus in the example for the application of the calculator, mathematical operations are performed on numbers. When a fuzz tester do mathematical operations on alphabets or simply purges on the keyboard, then does application resists with the malformed input.

Fuzz testing is a low effort sort of testing. In fuzz testing, the tester inputs the malformed data into the application and undergoes different test cases and sooner or later it might cause a crash. As the crashes are met, the software vendor spot the programming error and these errors are fixed. Hence improves the quality of software.

1.4 Fuzz Testing

Fuzz testing is one of the software testing technique which is used to discover errors and security loopholes in code of software, operating systems when we input random data into the system and make a crash to the system. Fuzz testing started in 1989 and was developed at the University of Wisconsin by Barton Wisconsin. Fuzzers works for problems that may cause program to crash, example buffer overflow, denial of service attacks, cross-site scripting, SQL injection and format bugs. Malicious users

take advantage of such crashes and enter into the system and takes over the system in least possible time. Spyware, worms and key loggers, fuzz testing here is less effective as it does not cause program to crash. Fuzz testing reveals vulnerabilities when software is debugged. Fuzz testing helps in exploitation of the software.

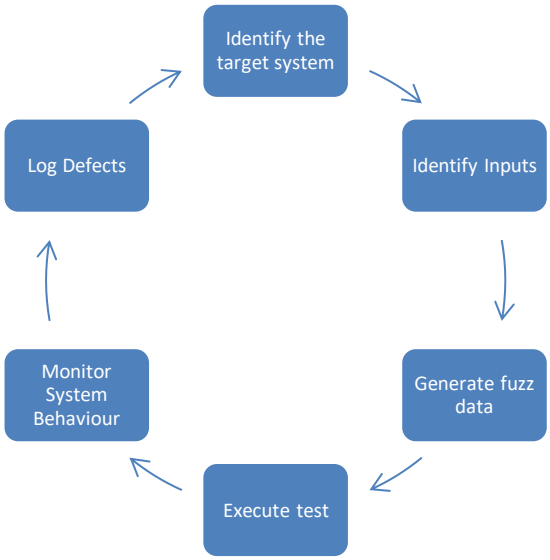

Figure 1.1 Stages of Fuzz Testing

1.4.1 Types of Fuzzers

The categories of fuzzers are: inputs from scratch, input structure and program structure. Under these categories all types of fuzzers are explained.

1.4.1.1 Aware From Inputs From Scratch or Modifying Inputs

When the inputs are from scratch or inputs are modified, the fuzzers are classified into:

1.4.1.1.1 Mutation-Based Buffer

A mutation-based fuzzer, generates the inputs by modifying (or mutating)the seeds. For example, user provides a set of image files as input and mutation-based fuzzer modifies the seeds which then produces semi-valid variations of each seed. At initial stage, files may contain thousands of inputs which are similar. To automate seed selection, the best seeds are picked so that maximizes total number of bugs found during a fuzzing.

1.4.1.1.2 Generation-Based Buffer

A generation-based fuzzer, inputs are generated from the scratch. This type of fuzzer generally takes the input model which is provided by the user to generate new inputs. This fuzzer usually does not depend on the existence or quality of set of inputs provided. There are fuzzers that have the capability to do both, one that generates input from scratch and second fuzzer which generates input by mutation of existant seeds.

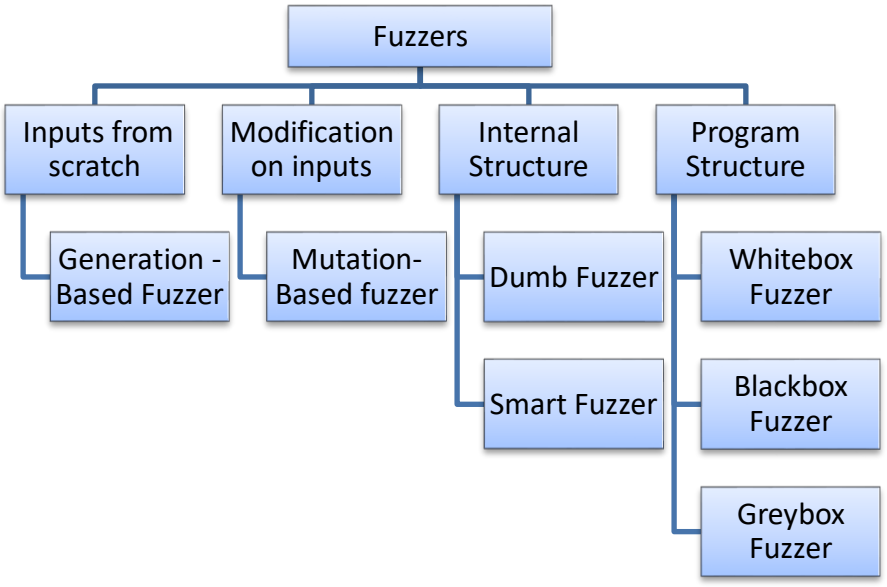

Figure 1.2 Categories of fuzzer

1.4.1.2 Aware of Input structure

In this type of fuzzer, structured inputs like file, sequence of messages, keyboard or mouse events are generated. These fuzzers are used for testing the structure which the input into: valid input which is accepted and the invalid input which is rejected when program is processed. Valid inputs are specified in an input model such formal grammars, GUI-models, file formats and network model. Databases, environment variables, shared memory or interleaving of threads are not considered as inputs.

1.4.1.2.1 Smart Fuzzer

A smart fuzzer is a model-based, protocol-based or grammar based fuzzers which generates valid inputs. For example, inputs can be modeled to abstract syntax tree which would employ random transformation on moving complete subtree from node to node using the smart-mutation based fuzzer. In a generation based fuzzer, for instance if the input is modeled to produce the rules that would generate inputs which are valid according to the grammar.

1.4.1.2.2 Dumb Fuzzer

In a dumb fuzzer, requirement for the input model is not known and so variety of programs can be fuzzed. For example, one of the dumb mutation-based fuzzer is AFL which modifies the file simply flipping random bits with values and by either deleting or moving blocks of data. Dumb fuzzer stresses on the parse code and not the main components of program and generates a small set of valid input.

1.4.1.3 Aware of Program Structure

When high degree of code coverage is achieved, working of fuzzer is considered to be effective. If the elements in structure of the program is not considered by the fuzzer, then the bugs in the program stays hidden and not exploited. For example division by zero error caused by the division operator or when a program is crashed

by a system call, if these elements are not considered by the fuzzers, then they are critical to the software or system.

1.4.1.3.1 Blackbox fuzzer

Blackbox fuzzer is not aware of internal structure of the program and therefore considers the program as a black box. For example, generation of inputs randomly using a random testing tool can be taken into account as a blackbox fuzzer. Blackbox fuzzer run the programs of any size and do parallelization by running hundreds of inputs per seconds.

1.4.1.3.2 Whitebox Fuzzer

A whitebox fuzzer works in a systematic order which increases the code coverage or find the critical locations of program. For example, SAGE is whitebox fuzzer which do symbolic execution and explores different paths in program. For exploitation of bugs, whitebox fuzzer are very effective. The disadvantage with whitebox fuzzer is that it takes relative longer time period for input generation than blackbox fuzzer.

1.4.1.3.3 Greybox Fuzzer

A greybox fuzzer works on instrumentation about the program rather than analysing it. For example, if to go through the basic block transition then greybox fuzzers are used such as AFL, libFuzzer. However performance overhead increases but the code coverage which is necessary during fuzzing increases thus making greybox fuzzers efficient tools for finding vulnerabilities.

1.5 Vulnerability and Exploit

1.5.1 Vulnerability

Vulnerability is defined as the weakness of the system that allows the malicious user to enter into the system and affects the information assurance of the system. Vulnerability comprises of three of three sections as shown in figure 3.

1.5.2 Exploit

An exploit consists of a well-defined data, piece of software or some commands which takes advantage of vulnerability or bug so that attacker can cause the unexpected behaviour to the system and gains control over the system.

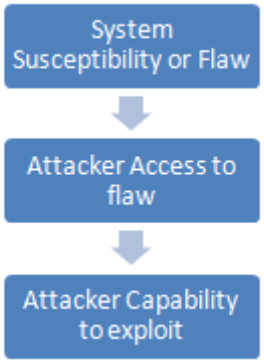

Figure 1.3 Elements in Vulnerabilities

1.6 Stack Based Buffer Overflow

Buffer overflow in stack (stack buffer overrun) occurs when the program is written to a memory address present on stack when there is program's call. Buffer in a stack is fixed length. Buffer overflow in stack occurs when size of the buffer is fixed and more data written as per the defined size of the buffer located on stack. The data adjacent to the stack results in corruption and with the overflow, causes the target program to crash or to operate incorrectly. Since there are active function calls on stack, buffer in stack causes overflow which derails execution.

Stack buffer overflow are caused deliberately by an attack known as stack smashing. If the affected program is running with some privileges, or data are accepted from untrusted network (example, a webserver) then there are more chances that the bug detected is a vulnerability which will affect the security of the system. The buffer in stack when overflows with data provided by the malicious user, corrupts the stack by

injection of executable code into the program and hence takes control over the process. Stack buffer overflow is one of the oldest vulnerability which allows the untrusted users to gain access into the system.

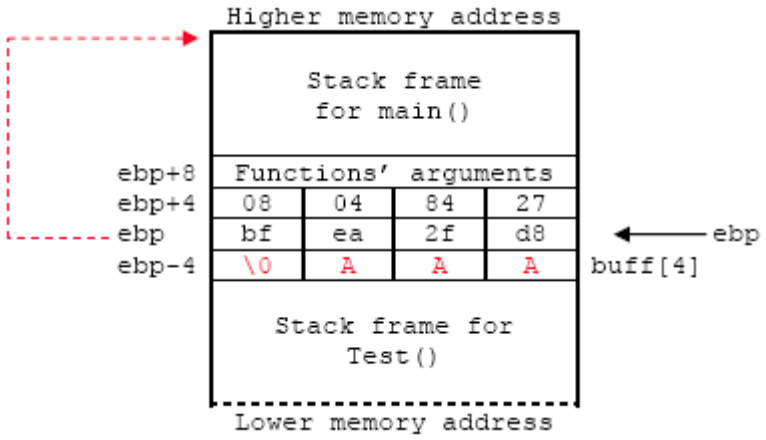

Figure 1.4 Buffer Overflow in Stack

Chapter 2

Literature Survey

Testing a software involves examining the behaviour of the system so that faults can be discovered. The inputs provided to the system, challenges regarding correct behaviour of the system from incorrect behaviour which is called test oracle problem [1]. The automated input generation have been both subject of search based technique and symbolic execution [1][7]. Software testing aims to assure the quality of software under test. This quality is improved by regression testing, test-case prioritization so that to execute the test cases in order [2]. Now for improving safety and reliability of software one of the testing techniques being used is fuzz testing. Fuzz testing is a technique for finding out the vulnerabilities and defects by sending malformed and unexpected inputs to software [3]. For example fuzzing is appropriate for medical device manufacturers, computing and network equipment manufacturers, healthcare delivery organizations, researchers and organizers that live in medical industry supply chain [3]. Medical devices communicate with other smart medical equipment, and the protocols and file formats used different machines are excellent candidate for fuzzing [3].

In Application Programming Interface(API), fuzz testing is used to insert unexpected data into the parameters of functions and to monitor for resulting program errors or exceptions in order to test the security of APIs [5]. This paper proposed methodology to automatically find paths between inputs of program and faulty APIs [5]. SecFuzz is a fuzz-testing security protocol fuzzer which generates valid inputs and using the set of fuzz protocols to mutate the inputs [6]. To mutate encrypted messages, fuzzers are provides with keys and cryptographic algorithm [6]. Fuzz testing can be adopted on a Finite State Machine(FSM) model and this algorithm is efficient for state space reduction [4].

There are certain bugs which are not threat to security of the system, thus only causing program to crash when unexpected data is input to program which provides

wrong output, hence theses bugs are not exploitable. Vulnerability is defined when a bug is exploitable. Attackers uses such vulnerability and implement code into the program which performs unexpected behaviour and hack into the system which affects the security of the system [3][10].

Inputs in test case which behaves according to the program where no vulnerabilites could be found, avoids the program to crash and thus hackers may control to the program and perform malicious tasks. Such test cases are called exploits. If any unexpected data is input by the programmer vulnerability regarding the program is found and hence counter measures are performed [10].

Stack based buffer overflow is well-known software vulnerability. These are considered one of the serious threats to the computer system. Automated method for exploit generation is there which constructs exploits for stack buffer overflow and prioritize software bugs [9]. Thus requires method for detecting different classes of vulnerabilities which is smart fuzzing [8]. A overflow in buffer to a subroutine is to break into system and to cause a security attack. When we write the data into a buffer it overflows the boundary and thus corrupted data overwrites the valid data. For example strcpy(), memcpy(), gets() [9][10]. Fuzzers uses the concolic execution to analyze target program [8]. Concolic testing is a strategy combining the accuracy of concrete execution and symbolic execution. It explores only one path at a time. So first it executes with concrete random inputs and then for branch condition using symbolic execution [7]. By giving unexpected inputs to the system we can find the vulnerability and do the exploitation which stops the hackers to hack into the system [16].

For exploit generation, stack buffer overflow exploitation it is necessary to prioritize bugs. Bugs and vulnerabilities can be detected at binary code and source code analysis [9][14]. Binary code analysis is a preferable approach because abstractions of high-level languages hides specification of program operations that helps for detecting bugs and evaluates the severity [9]. So bug detection technique is based on symbolic execution [9][12][13].

2.1 CPU Model

Central Processing Unit (CPU) is an electronic circuit in a computer that carries instruction of a program and perform arithmetic, logical, control and I/O operations as per instructions. Simply, its starts with fetching the instruction, decodes the instruction i.e. instruction is converted into signals which control other parts of CPU [17][18]. In decoding step, there is a question of interpretation that if in the memory is it instruction or data transferred. With final step , execution of instruction.

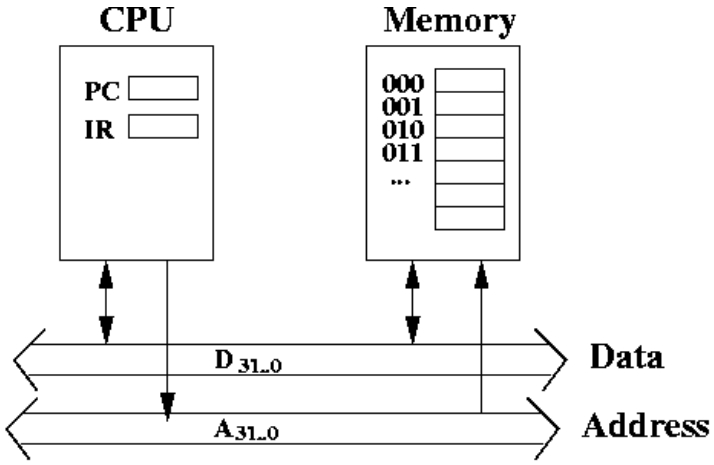

Figure 2.1 CPU Model [1]

2.2 Stack Model

Stack is a memory region and structure for function call. Stack woks on the principle of LIFO (Last In First Out). The registers used in stack are:

i. EIP register: Extended Instruction Pointer, points to next instruction to be executed
ii. ESP register: Extended Stack Pointer, points to top of stack
iii. EBP register: Extended Base Pointer, points to location of current stack frame

Operations performed in stack are :

i. NOP: No operation
ii. PUSH: Push the item to top of stack
iii. POP: Remove the item from top of stack

The operation with registers performed are:

i. NOP: No operations
ii. %eax MOV: Store value to register
iii. %edx ADDB: Add bit value to memory address pointed by register
iv. %ebp PUSH: Push the base-pointer onto stack. The sub operations performed are SUB 4, %esp and MOV %ebp, %esp
v. %ebp POP: Pop base pointer from stack. The sub operations performed are MOV %esp, %edp and ADD 4, %esp
vi. JMP: Continue at address
vii. CMP: Set zero flag if equal
viii. JNE: Jump if zero flag not set
ix. CALL 0x804e47c: Calls subroutine
x. RET: Return from subroutine

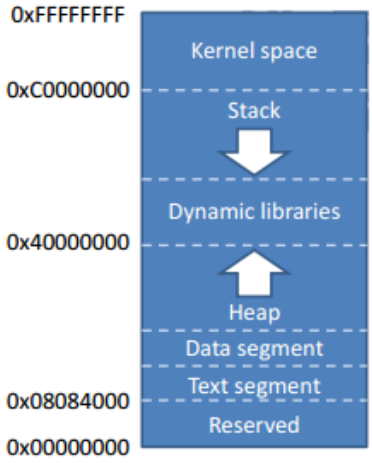

Figure 2.2 Memory layout

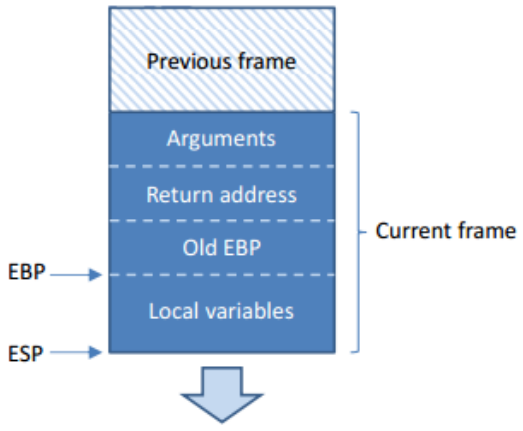

Figure 2.3 Stack Frame

2.3 Symbolic Execution

Symbolic execution is one of the popular technique of software testing and was proposed in lasts of 1970s for software testing. In this execution all the paths of the program are explored. The specific values are replaced with symbolic values [9]. The value range of variables that are represented by symbolic expressions can have any value when program runs initially. With the execution of program, the non- symbolic variables will be tainted by symbolic variables and hence the value range will be restricted [7].

Example code:

```
1 void T(int a)
2 {
3       int b=0;
4       if(a>=0)
5           printf("a is greater than or equal to 0");
6       else
7           printf("a is less than 0");
8       b = a+100;
9       if(b == 2011)
10          printf("b is equal to 2011);
11      else
12          printf("b is not equal to 2011);
13 }
```

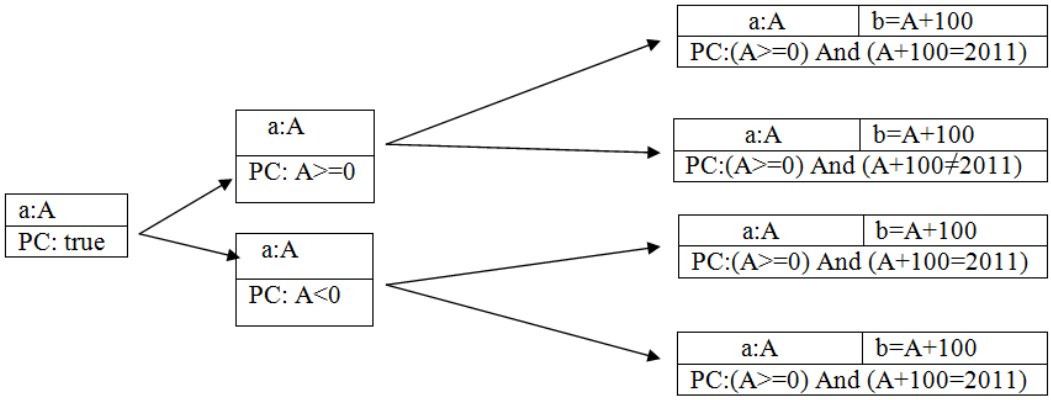

Figure 2.4: Symbolic Execution Tree [7]

From above example and in figure 2.4, variable 'a' is replaced by symbol 'A' when program starts to run. At line 4 execution is forked and hence there are two constraints A >= 0 and A < 0. The concrete value 'b' is tainted by variable 'a' at line 8 i.e. b= a+100, therefore b becomes symbolic and its value is A+100. At line 9 again the execution is being forked as variable 'b' is symbolic and hence again divided into two constraints A≠ 2011 and A = 2011[19]. The path where the constraint is (A>=0) And (A+100=2011) is dropped because this path is an infeasible path.

Symbolic execution has explored all four paths but since the third path is infeasible for the three paths are explored, each path is passed to constraint solver to get solution.

2.4 Concolic Execution

Concolic execution(concrete execution + symbolic execution) which calculates the path and vulnerability constraints for all execution path in the program [8]. Concolic testing explores only one path at a time and tests the program concretely and symbolically. It initially tests with the random input and then symbolic execution collects all the branch conditions.

Fuzzing method uses concolic execution technique for detecting vulnerabilities such as stack based buffer overflow. In this vulnerability constraint, each path is calculated by analysis of binary code and combines with path constrain in order to generate appropriate test data [7]. From above example code:

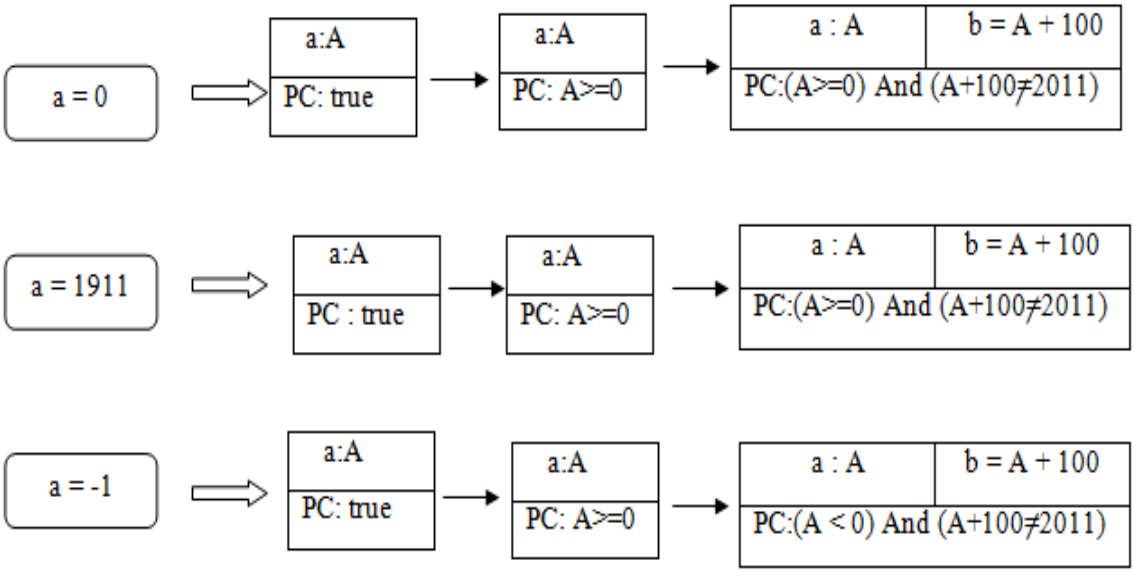

Figure 2.5 Concolic testing [7]

2.5 Vulnerability and Exploit

There are certain bugs which are not threat to security of the system, thus only causing program to crash when unexpected data is input to program which provides wrong output, hence theses are not exploitable. Vulnerability is defined when a bug is exploitable. Attackers uses such vulnerability and implement code into the program which performs malicious behaviour and hack into the system which affects the security of the system.

When in a test case inputs which behave according to the program , no vulnerability could be found, avoids the program to crash and hackers can have control to the program and perform malicious tasks. Such test cases are called exploits. If any

unexpected data is input by the programmer, vulnerability regarding the program is found and hence counter measures are performed.

Exploits often contains piece of binary code as payload called shellcode which performs malicious tasks [8].

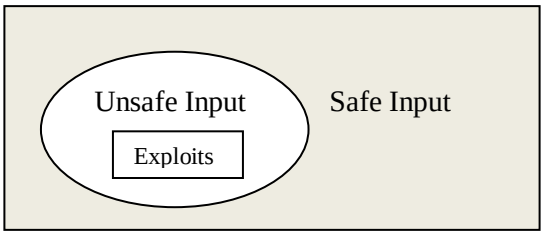

Figure 2.6 Relationship between exploit and input space [7]

2.6 Fuzz testing

For finding bugs in a software, fuzz testing or fuzzing is one of the testing techniques [26]. Since the bugs are security related so fuzz testing on these bugs are against interface of programs. Fuzzing does not ascertain completeness or correctness of program instead it focuses on identification of anomalies in an application by applying some protocols, attack heuristics [30].

Fuzzing process commonly used by companies and open source projects so that the quality of the software can be improved by the analysts who discovers the vulnerabilities and reports bugs [30]. In comaprison with other sofware testing techniques, reverse engineering and source code audits, fuzzing is found to be more effective and cost efficient. Comparison done with traditional software testing techniques or even source code audits and reverse engineering, fuzzing is found to be more effective and cost efficient [31].

Biochiang Cui *et al* [22], this paper describes API in-memory fuzz testing technique which uses dynamic symbolic execution for locating routines and instruction belonging to target binary executable. Input data is parsed and processed. In this

method during testing stage, loops are constructed using binary instrumentation. The model proposed use to mutate the taint memory values present in each loop and construction of each loop for routines is performed by binary instrumentation. According to experiments performed, the designed model effectively detects vulnerabilities such as buffer overflows. Comparison done with traditional fuzzing tools, this proposed model eliminates the interruption of executed path and 95% execution speed is enhanced .

M Mouzarani *et al.* [23], this paper presented a new model of fuzzer for detecting heap-based buffer overflow based on concolic execution. The model proposed executes the concrete data of target program and the constraints of each path obtained are executed symbolically. The path constraints obtained generates test data that traverses new execution path in target program. Heap buffer overflow was calculated for each executed path. The constraints obtained determines which input data have caused overflow in the executed path. This model again repeats the cycle by generating new test data which is obtained by combination of path and vulnerability constraints, which traverses path and again specifies vulnerabilities in the path. The model tested different group of programs and results obtained shows vulnerabilities for programs accurately.

Chen *et al.* [24], This paper created a model MiBO for detecting vulnerabilities due to buffer overflow caused by malicious inputs. For identification of MiBO vulnerabilities, white-box testing technique was used to analyze all execution paths. Black-box testing technique triggered MiBO vulnerabilities by providing different inputs, but limited coverage was achieved. When the vulnerabilities were identified i.e. there was a 'hit', thus crash happened by the test case input. This paper presented the model which identifies non-crash MiBO vulnerabilities using white-box testing technique. Without source code, dynamically discovers likely memory layouts to help the fuzzing process. This is very challenging since memory addresses and layouts keep changing with the running of software. In different executions with different inputs, the layouts may also change. To address these challenges, we

selectively analyze memory operations to identify memory layouts. If a buffer border identified from the memory layout is exceeded, an error will be reported. The fuzzing results will be compared with the layout for future input generation, which greatly increases the opportunity to expose MiBO vulnerabilities. This paper implemented a prototype called ArtFuzz and performed several evaluations. ArtFuzz discovered 23 real MiBO vulnerabilities (including 8 zero-day MiBO vulnerabilities) in nine applications.

Kim *et al*. [25], this paper uses the symbolic execution technique to analyze vulnerabilities in code. As business are becoming more dependent on software and computer programs so there is need for elimination of security bugs. This paper represents smart fuzzing which uses data from symbolic execution engine which automatically generates input which causes the crash. Symbolic execution engine being used generates the data automatically that are against vulnerability issues. This paper also represents a method for user in clarifying error reports because some verification tools fails to verify the program and hence provides the wrong result, so users manually verifies the program which becomes time-consuming, and error-prone tasks. This technique computes small queries to user. This proved advantage in program inspection and debugging.

Table 2.1 Examples of Fuzzer

S.No	Fuzzer	Description
1	Taming compiler Fuzzer	Aggressive random testing tools which explores compiler bugs. More than 1700 bugs were found using a single test-case. Using ad hoc methods, undesirable test cases are filtered out. When large number of test cases were triggered, test cases were ordered to priority [32].

2	SNOOZE	SNOOZE is a network protocol fuzzer tool. Identifies security flaws in network protocol. This fuzzers allows the tester to operate on stateful operation of a protocol and generate messages for each state and focus on vulnerability classes. SIP protocol was implemented as initial prototype to test programs. This tool exposed real world bugs in the programs analyzed [33].
3	VUzzer	VUzzer is an application fuzzer that does not require any prior knowledge of application or input format. Static and dynamic analysis on applications are performed so there is maximum coverage and deeper paths can be explored. This fuzzing strategy evaluated three datasets: DARPA Grand Challenge binaries (CGC), real-world applications , and LAVA dataset. On all of these datasets, VUzzer yields significantly better results than other fuzzers [35].

2.7 Software Vulnerabilities

2.7.1 Stack Based Buffer Overflow

Stack buffer overflow is a common vulnerability and usually caused by unsafe functions, such as strcpy() and gets(). The return address of functions is stored in the stack frame and restored to EIP register when function returns. If the length of source input is not checked, stack smashing will happen by feeding a long input over the boundary of the destination local buffer[5]. The corrupted return address will cause control flow hijacked when the function returns.

For the exploitation of buffer overflow the pre-requisite is to gather knowledge about memory. Goal is to see how memory and execution of instructions are performed by CPU. The technique stack-based buffer overflow used involved how the variable is

overfilled on memory stack of program and overwrites the adjacent memory locations.

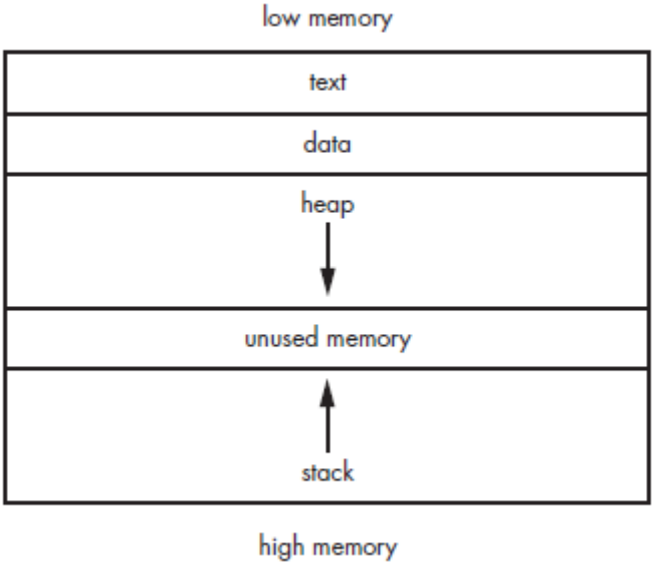

Figure 2.7 Memory Address in Stack Frame

At lower address, text segment which consists of program code which is executed and data segment which consists of global information regarding the program. At runtime, portions of stack and heap are shared which are present at high address. The size of the stack is fixed and it stores function arguments, local variables. Heap holds dynamic variables. The stack is increased when functions or subroutines are called and at lower addresses the top of the stack points[17]. In CPU we have general-purpose registers which stores data for future use. These include: EIP, ESP, EBP, ESI , EDI , EAX and EBX, ECX, EDX .

In figure 2.8, ESP points at low memory address and is a top of the stack frame and at the bottom of the stack frame EBP points at the high memory address. Instruction pointer holds the address of the next instruction to be executed [10]. Because goal is to control execution by delivering malicious code and make the target machine execute according to the unexpected user, with control over EIP target machine can be controlled.

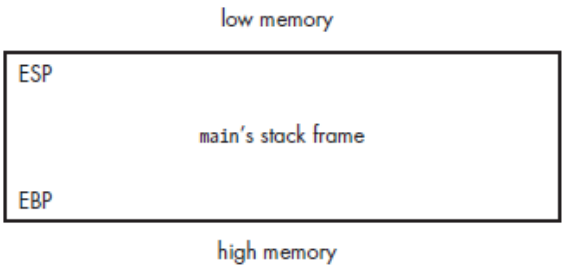

Figure 2.8: Stack frame layout

The stack is LIFO data structure. With the execution of program function, in a stack frame information is pushed onto stack. Once execution of function finishes, stack pointer and base pointer points back to the stack frame of caller function and where the execution was left in the caller function it continues. However, the CPU knows the location in the memory where to continue and thus obtain its information from return address, which is then pushed onto stack when a function is called. The function main of a program is allocated to stack frame[5]. Main calls the other functions, before the other functions are pushed into the stack frame, main keeps track where the execution to be continued when other function returns. After function finishes, it returns, and restore the execution to main when the return address is loaded in instruction pointer register.

Maryam Mouzarani *et al.* [8], they have introduced the method for detecting vulnerabilities is smart fuzzing. This technique uses concolic execution which detects buffer overflow vulnerability. For each execution path, buffer overflow constraints and path are symbolically calculated. As the calculation of the vulnerability constraints are been done they are stored in stack memory. The vulnerability constraint generated determines the length of input data up to which the overflow in the stack buffer occurs. Estimation of length and address of variables in stack has been considered by the structure of stack and accessing the local variables in binary codes.

Buffer overflow constraints have been calculated on the basis of estimated length and addresses. To find the overflow in stack buffer, the vulnerability constraints

considers which part of input data involves in overflow and thus helped in fuzzing. The results of this estimation has calculated stack overflow constraints.

Szekeres *et al.* [21], this paper describes the systemization in memory corruption and exploitation models. Classification of the defense techniques is based on the exploit mitigation and inhibition of exploit. Using the evaluation criteria such as robustness, performance and compatibility they have developed a new defense software. The model developed takes into consideration memory corruption attacks and identification of different security policies. Clarification of attack vectors which are left unprotected by currently and previously designed protections by matching their policies with different exploits. Performance, compatibility and robustness are evaluated and compared.

Chen *et al*. [26], this paper has proposed a model AURORA which is a robust kernel-based solution, to security problem which controls buffer overflow attacks. The model utilizes either addresses of buffer with stores input strings or signature which detects and blocks buffer overflow attacks in kernel. The model creates the signatures based on the buffer overflow attacks so there is no need to create new signatures for instances of new attacks. When the process is under buffer overflow attack, the model allows the execution or termination without the cost of repeated crashes or idleness of process, thus the model is robust to control buffer overflow attacks. There is no need for modification of source code and furthermore model is compatible with application and operating system. This model has shown less than 1 % overhead and least false positives and hence accurately blocks various buffer overflow attacks.

Wang *et al*. [27], this paper has proposed a model called Runtime Integer Checking via Buffer Overflow-RICB model describing integer overflow vulnerability which causes buffer overflow. This models decompiles execute file to assembly language, debugging of the execute file, checks overflow points and thus checks buffer overflow which occurs due to integer overflow. This model has implemented its

approach in stack overflow, heap overflow and format string overflow. The approach implemented resulted in effectiveness and efficiency of these overflows. Experiments based on the model are effective and efficient.

Sidiroglou *et al.* [28], this paper is based on the hypothesis that each code function is treated as a transaction which aborts whenever attack is detected and does not affect the execution of an application. For direct tradeoff between security and performance, this mechanism enables or disable components in response of external events. This paper has implemented a stand-alone tool DYBOC which finds number of vulnerable application. The performance indicates 20% slow-down for Apache when in full protection and with selective protection the percentage goes to 1.5%. The transaction hypothesis were validated via two experiments : first one was that the model was applied to 17 vulnerable application among which 14 were fixed and second was Apache behaviour was examined with 154 vulnerable routines, result was 139 cases approximately 90% resulted in correct behaviour.

2.7.2 Off-by-one Overflow

Off-by-one error is also a common bug and arises from an error boundary condition. If EBP register or a pointer variable is overwritten, it will be exploitable [17]. Because EBP register will update ESP register when the function returns, and the corrupted pointer may write data to arbitrary locations, both cases could influence control flow indirectly.

2.7.3 Heap Buffer Overflow

A chunk will try to merge adjacent free chunks when it is deallocated. If a heap buffer overflow vulnerability occurs, attackers can overwrite the header of next chunk to fake the size and the value of pointers [16]. When the current chunk is deallocated, the allocator will try to merge with next chunk and the unlink operation will cause arbitrary write of 4-byte data by attackers to arbitrary memory addresses.

2.7.4 Uninitialized Variable

The function moves ESP register to deallocate local variables when functions return, and the prologue increases the size of the local variables to ESP register when calling a function. If a local variable without initialization is used, its value will reuse the value of previous function invocation. When invoking two function such as a() and b() sequentially, the stack frame of the last function will reuse the overlapping space with the previous functions [20]. Therefore, if the previous variable can be controlled, the current variable can also be controlled via this vulnerability.

2.7.5 Format String

Some C functions that perform formatting output, such as printf(), use unchecked input as the format string parameter, attackers can use some format tokens, such as %x, %s, and %n, to print the information in stack or write data to arbitrary memory addresses [11]. When format tokens are encountered in a format string, the program expects that the data are retrieved from the stack, but if the input is not provided in function arguments, the program will read from or write to wrong addresses in the stack. The %n format token writes the number of characters output in front of itself to the location provided in arguments, and attackers can use %n format token to write arbitrary value to arbitrary address.

2.8 Protection Mechanism

Prevention from attacks, many protection mechanisms have been proposed and applied in some compilers or operating systems. Those methods aim to increase the difficulties of successful attacks. The main mechanisms are listed below:

2.8.1 ASLR (Address Space Layout Randomisation)

ASLR randomizes the locations of memory regions, so that attacks won't work because the address of shellcode is random and unexpected. In Linux, ASLR randomizes stack, heap and share library, but not for the program image [17].

2.8.2 W⊕X (Writable⊕eXecutable)

W⊕X sets a memory region either writable or executable. Exploits usually contain shellcode as payload, but W⊕X stops the shellcode from being "executed" because the exploit must be "written" to memory [17].

2.8.3 Stack Hardening

In order to protect stack against buffer overflow attacks, compilers have implemented some techniques to defend return addresses from corruption. In GCC compiler, StackGuard inserts "canary" in front of return addresses and checks whether the value is changed when functions return. In addition, Stackshield copies away return addresses to a non-overflowable area and restores it when functions return [17].

2.8.4 Heap Hardening

To protect heap from smashing, heap consistency checking has implemented to check whether the metadata, which records the information about neighboring chunks, is corrupted or not, whenever a heap block is freed. In addition, many safe functions are provided for programmers to avoid using unsafe functions. For example, strcpy() function is very dangerous because of buffer smashing, and strncpy() function is the safe version of strcpy() function that copies a string with bound checking [17].

2.10 Research Gaps

This section tells about the gaps encountered during literature survey in the area of stack buffer overflow and fuzz testing.

1. Fuzzers are not capable to evaluate path constraints for data which are inputs from file and keyboard as these inputs are tainted [8] .
2. Vulnerability constraints of copied data are not evaluated by fuzzers and hence does not assists in exploitation of program [23].
3. Analysis of target program is not done by MiniFuzz fuzzer and neither input data length is extended [35] .
4. For detecting buffer overflow vulnerabilities high code and path coverage is required and hence fuzzers fails to do such or execution time is high [4][20] .

2.11 Problem Formulation

As new projects comes up, so does the security threats into the system. There are some random tools which exploits such threats. Such tools are fuzz tools like SPIKE, PROTOS which exploited the threats in the system. Fuzzing is effective because it fills the void or gap that other testing techniques does not. Buffer overflow is one of the common software vulnerabilities. AFL fuzzer adds instructions to the code which detects the code paths. It is an instrumentation-guided fuzzer which is capable of synthesizing complex files in wide range of target programs, thus lessens the need purpose-built, syntax-aware tools. Afl fuzzer is a unique crash explorer, a test case minimizer, a fault-triggering allocator and a syntax analyzer which makes evaluation of crashing bugs simple. On every execution path exploitation of vulnerabilities occurs. In comparison to other fuzzer tools [4][20][33][34], AFL is more efficient and is one of the best fuzzing tools available.

2.12 Objectives

1. To study and analysis the work done in area of fuzz testing

2. To propose a stack based buffer overflow model to identify maximum number of exploits

3. To implement and validate the proposed model for number of vulnerabilities and to resolve stack based buffer overflow.

2.13 Overview

The structure of thesis is as follows.

1. Chapter 2: The literature survey of stack based buffer overflow, counter measure and fuzz testing with tools being used.
2. Chapter 3: Description and designing of proposed model.
3. Chapter 4: Implementation of the proposed model and results obtained.
4. Chapter 5: Conclusion after analyzing the results and future work.

Chapter 3

Modelling and Designing of

Stack based-American Fuzzy Lop (SAFAL)

This chapter describes the modelling and design of the SAFAL model. It gives details regarding the workflow of proposed model and steps necessary for the working of model. This chapters describes the activity diagram and class diagram of SAFAL model.

3.1 Workflow of Model

Fuzzing is powerful software testing technique for identification of security issued in the software. Bugs such as remote code execution, privilege escalation are found which are critical for software.

American Fuzzy Lop(AFL) is a simple and rock-solid instrumentation-guided genetic algorithm.

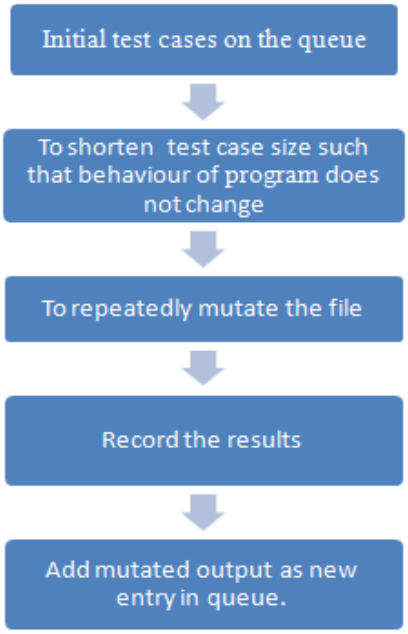

Figure 3.1 Flowchart of SAFAL model

These steps are performed on a software pdfcrack using AFL fuzzing technique which find the vulnerabilitiy in the software. High-level of code coverage is there. This strategy causes afl-fuzz to reach a high level of code coverage for its testing. The fuzzer starts detecting when a new path is being triggered and more input files are created and identification of bugs started.

3.2 Binary Instrumentation

With the availability of source code, the instrumentation is being carried out by the tool which functions as a substitute for gcc or clang in any third-party code compilation process. By using afl-fuzz, most programs produce the fastest results.
If the source code is not available or if the source program recompilation of the destination program cannot/does not want to be done, AFL supports QEMU mode. This is done when the target program could not be built with afl-gcc.

3.3 Choosing initial test cases

For the initial test cases, there is requirement of one or more input files that contains data which is expected by target application for the fuzzer to operate correctly. If the large corpus of data is there for screening, then use of afl-cmin utility. This afl-utility identifies the subset of files. Different code paths originates in target binary for these files.

3.4 Fuzzing binaries

The process of fuzzing is performed by the model afl-fuzz utility. Fuzzing binaries with the test cases requires read-only directory, to store the findings requires a separate place and the path to which binary can be tested.
./afl-fuzz -i input -o output /path/to/program @@
Programs which receives input from file uses '@@', marks location in the command line of target and to where to place the input file name.

3.5 Interpreting output

For interpreting the output, the fuzzer completes one queue, which may take some time. In the output directory, three subdirectories are created:

1. queue - In queue subdirectory, there are test cases for which every path is to be executed and the file is provided by the user. To shrink the size of the file afl-cmin tool can be used. This generally offer equivalent edge coverage for subset of files.

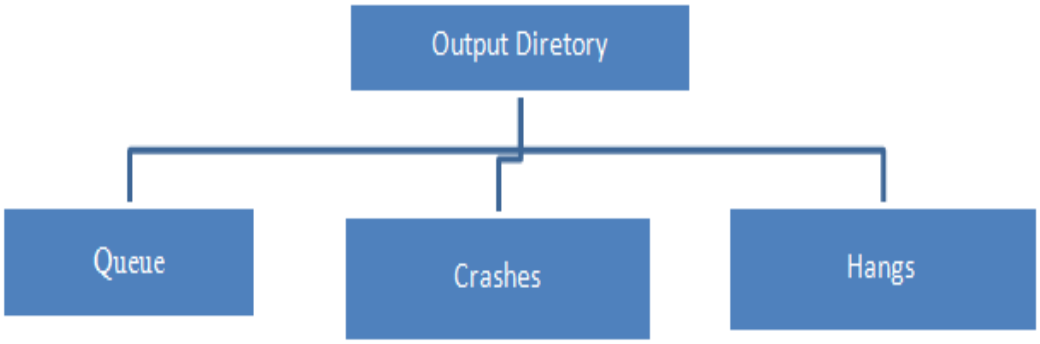

Figure 3.2 Subdirectories of Output Directory

2. crashes - In the tested program some of the test cases receives fatal signal such as SIGSEGV, SIGILL. So these crashes are stored in crash directroy.
3. hangs - There are test cases that causes the test cases of the program to time out. Before the test cases are hanged there is the default time limit which is larger of 1 second and value of -t parameter. If to change the value than AFL_HANG_TMOUT setting is to be done.

If there is state transition involved in the executed path and not previously recorded than crashed and hangs are considered to be 'unique'. The names of the files in crashes and hangs are related with parent, thus help in debugging.

3.6 Parallelized fuzzing

In afl-fuzz every instance is generally associated with one core i.e. on a multicore system, for full utilization of software parallelization is necessary. This parallel fuzzing mode provides interfacing AFL to others fuzzers and engines such as symbolic and concolic execution engine.

3.7 Fuzzer dictionaries

By default, mutation engine of AFL fuzzer performs optimization for compact data formats such as images, multimedia, compressed data, shell scripts. It is less suited for languages with most detailed and redundant verbiage, for example HTML, SQL, or JavaScript.

To avoid the chaos of building syntax tools, afl-fuzz provides a way to seed the fuzzing process with an optional dictionary of language keywords, magic headers, or other special tokens associated with the targeted data type with AFL.

3.9 Design of SAFAL Model

Software design is a process to transform the user's needs into a suitable way that helps the programmer in coding and software application. Software design is the first step in SDLC (Life Cycle Design Software), which shifts the concentration of the problem domain to the domain solution. It is about how SRS requirements can be met.

3.9.1 Activity diagram of SAFAL Model

The activity diagram shown in figure 3.3 is an important UML diagram that describes the dynamic characteristic of the system. Activity diagram is a flow chart which illustrate the flow from one activity to another. The activity can be depicted as a system operation. The flow may be in order, simultaneous, or branched. Activity diagrams with the nature of the flow control dealing with various elements such as

fork connection. The basic purpose of the activity diagrams is to capture the dynamic behavior of the system.

3.9.2 Class Diagram of SAFAL Model

The class diagram is a static diagram which characterizes the static view of an application. The class diagram used to visualize, describe and document various aspects of a system and also creates executable application code software. The class diagram figure 3.4 explains the attributes and operations of a class and system boundaries. Class diagrams in the modeling of object-oriented systems are widespread because they are the only UML diagrams that can be directly mapped to object-oriented languages.

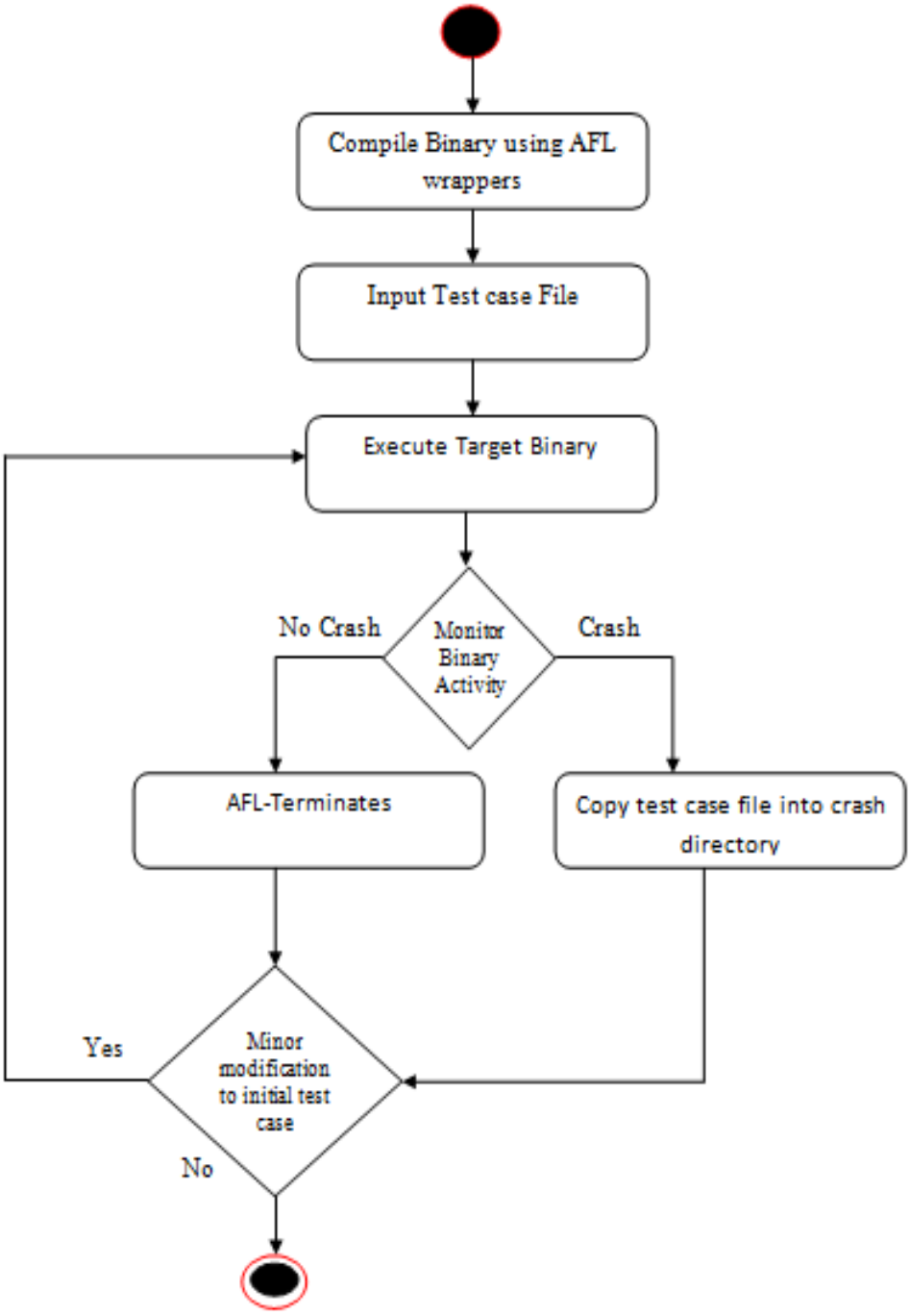

Figure 3.3 Activity Diagram of SAFAL Model

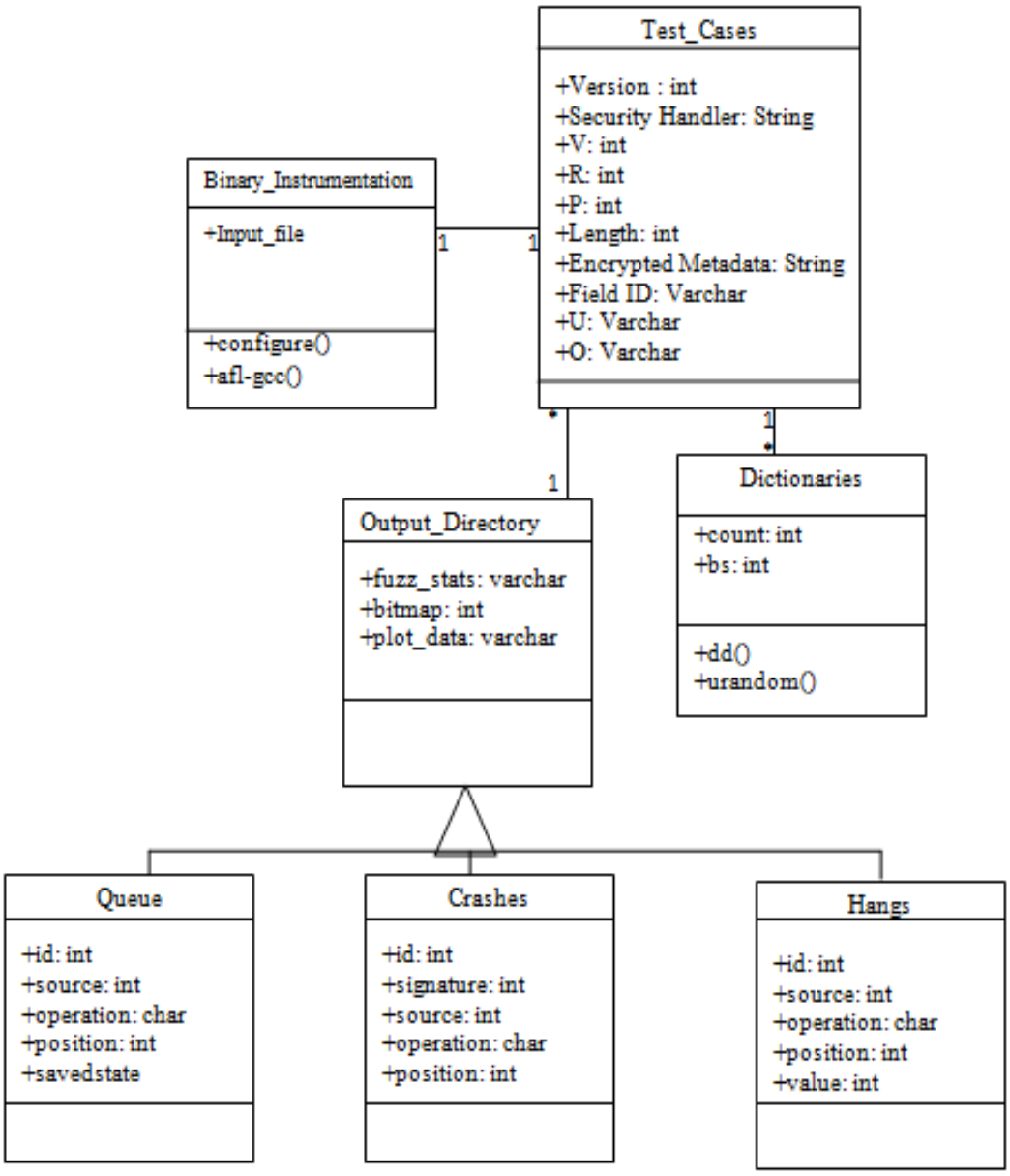

Figure 3.4 Class Diagram of SAFAL Model

Chapter 4

Implementation and Result Analysis

This chapter describes the implementation and results of SAFAL model. Implementation details includes the installation of software, the steps performed to get the results and snapshots of entire implementation.

4.1 Experimental Setup

4.1.1 Hardware and Software Used

Table 4.1 Hardware and Software Used

1.	Processor	Intel(R) Core(TM) i3 CPU
2.	RAM	6.00 GB
4.	Operating System	Windows 8.1 Pro
5.	Virtual Machine	VMware Workstation 12 Player
6.	Guest Operating System	Ubuntu-16.04.1
7.	Platform	AFL

4.2 Binary Instrumentation

American Fuzzy Lop (AFL) provides wrappers for gcc that adds instrumentation code. With the source code, keeping track of the binary execution path, the target binary uses the gcc wrappers. By using afl-fuzz, most programs produce the fastest results.

CXX=/usr/local/bin/afl-g++ ./configure

4.3 Initial Test Cases

For initial test cases, the input files are provided which have data as expected by the target application so that the fuzzer can operate correctly.

Table 4.2 Test case 1

PDF Version	1.3
Security Handler	Standard
V	1
R	2
P	-60
Length	40
Encrypted Metadata	True
FieldID	3ce86e902eba08848cedc21d64317f
U	F32ad8fda2ac3a46178937370a6f76071920499dad850a95f177b9089b6e20ca
O	68735ba82e3c37b5b8228510ba7e3ff6a57adff9901483de2357988c0a42aa48

Table 4.3 Test case 2

PDF Version	1.5
Security Handler	Standard
V	1
R	2
P	-4

Length	128
Encrypted Metadata	True
FieldID	Aa269ffec296b13f3ef835aaa13a0a08
U	C5fcd68fff42090ae5db77f3c6d7992e0122456a91bae5134273a6db134c87c4
O	36451bd39d753b7c1d10922c28e6665aa4f3353fb0348536893e3b1db5c579b

Table 4.4 Test case 3

PDF Version	1.4
Security Handler	Standard
V	2
R	3
P	-3104
Length	128
Encrypted Metadata	True
FieldID	66d36a3097e0f16f39955c6221e0c2a
U	2283eba99a7ec1b60a3a6fefc409941991f1dd5b6e5e0a88e5e0a7954217a8f78
O	0b519e6d22cff2512874511174dd17f0fd531ede8de0c811aa83ad6ba09358e9

4.4 Fuzzing Binaries

Fuzzing binaries with the test cases requires read-only directory, to store the findings requires a separate place and the path to which binary can be tested.

./afl-fuzz -i ~/i -o ~/o ./pdfcrack @@

Programs which takes input from file uses '@@', marks location in the command line of target and to where to place the input file name.

4.5 Execution of SAFAL Model

AFL fuzzer fires up and starts executing the program, AFL displays user interface with number of metrics.

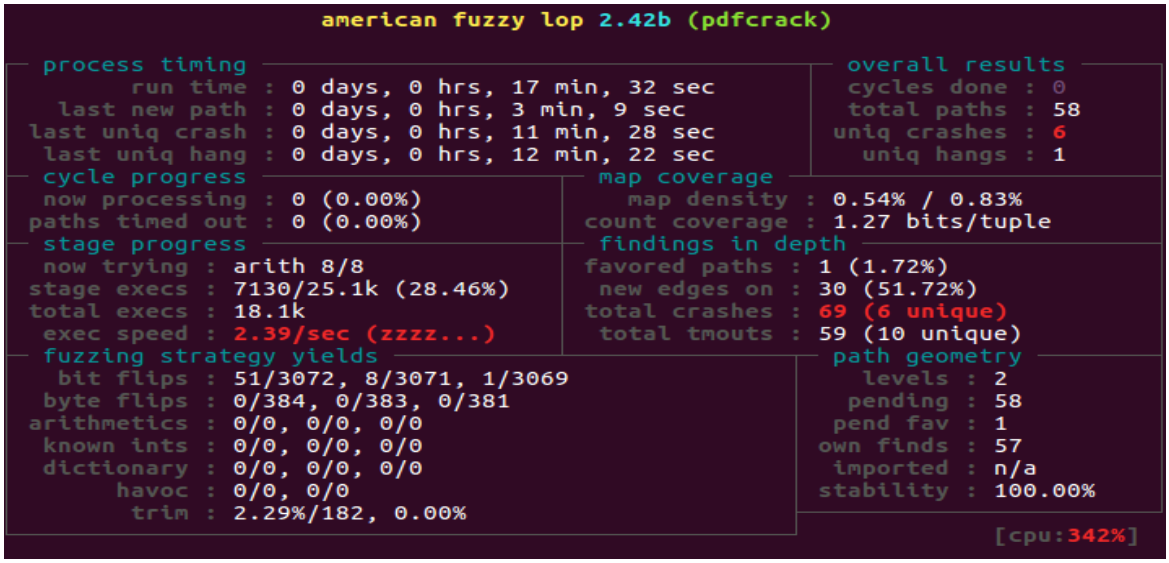

Figure 4.1 SAFAL User Interface with metrics

4.5.1 Processing Time

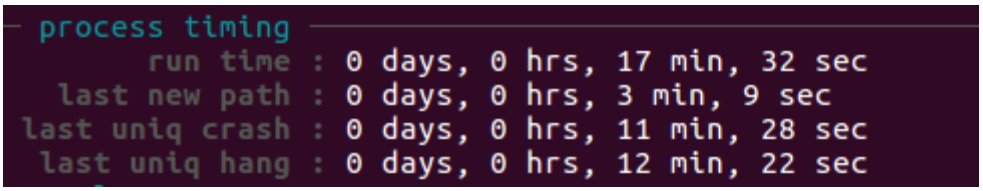

Figure 4.2 Processing Time

The processing timing describes the run time which specifies for how long the fuzzer runs. It also mentions the time at which new paths, unique crashes and hangs are explored. Every new crashes, paths and hangs are recorded in output directory.

4.5.2 Overall results

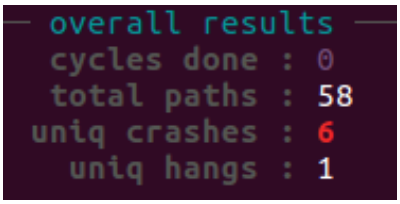

Figure 4.3 Overall Results

Overall results gives the description of number of times the test cases were discovered and then loop back to beginning. In figure 4.3, at this particular instance, total paths(cases) that were founds were 58 among which 6 were the unique crashes and unique hangs were 1. Test cases, hangs and crashes are recorded in output directory.

4.5.3 Map coverage

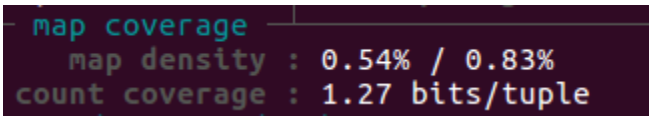

Figure 4.4 Map Coverage

Map coverage provides in and outs of the target binary. Map density tells about the number of branch tuples got hit and to proportion the bitmap can hold. Count coverage it throughs light on number of bits being covered for entire input corpus.

4.5.4 Findings in depth

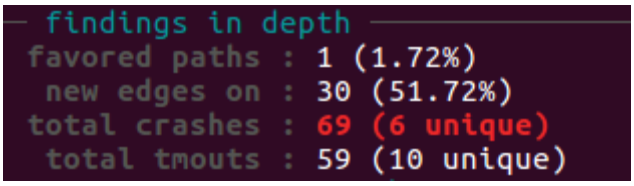

Figure 4.5 Findings in Depth

Findings in depth provides the details about the metrics that are of concern. It provides with information of the number of test cases with better edge coverage, total number of crashes and among them which crashes are at priority and the total timeouts.

4.5.5 Path geometry

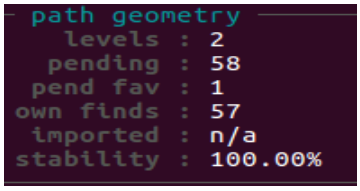

Figure 4.6 Path Geometry

In path geometry:

1. The first part is levels which keeps track of depth of path. The initial test cases are considered at "level 1". The derived test cases from fuzzing are considered at "level 2" and so forth.
2. Pending fav fields tells about the number of inputs of which fuzzing is yet to be done.
3. Favored field provides with statistics of number of entries want to enter the queue cycle and non-favored entries has to wait for couple of cycles.
4. Own finds tells about new paths found during fuzzing and if parallelized fuzzing process is being going than paths can be imported from other fuzzers.
5. Stability measures consistency of the data being fuzzed. If the behaviour of program same as input data than stability is 100%. When value goes down, purple color is shown which affects the fuzzer negatively.

4.5.6 Plot files

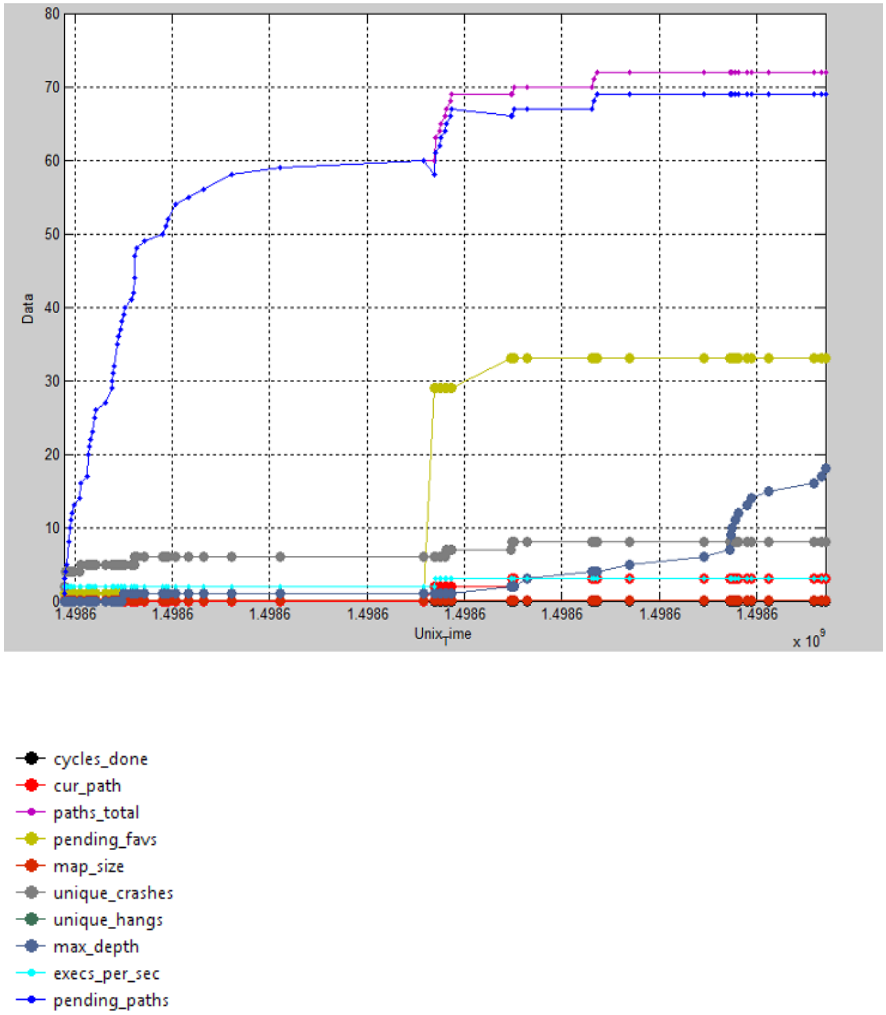

Figure 4.7 Resultant Graph

Figure 4.7 represents the resultant graph obtained from the fuzz statictics. After the execution of AFL fuzzer the statistics are:

Table 4.5 Fuzz Statistics

start_time	1498589941
last_update	1498593853
fuzzer_pid	101118
cycles_done	0
execs_done	158205

execs_per_sec	30.50
paths_total	72
paths_favored	36
paths_found	71
paths_imported	0
max_depth	3
cur_path	3
pending_favs	33
pending_total	69
variable_paths	0
bitmap_cvg	0.90%
unique_crashes	8
unique_hangs	18
last_path	1498592678
last_crash	1498592241
last_hang	1498593849
execs_since_crash	18511
exec_timeout	720
afl_banner	Pdfcrack
afl_version	2.42b
command_line	afl-fuzz -i /home/manisha/i2 -o /home/manisha/o ./pdfcrack -l @@

4.6 Crashes and Hangs

After the execution of the fuzzer, target binary is crashed and the more times the crashes occurs there is potential for identification of flaws. The crashes are stored in output directory in a file named crashes. For each crash, the operation, position, source file at which crash occurs are stored.

Table 4.6 Crash Exploration

Id	Signature	Source File	Operation	Position
000000	11	000000	Filp1	12
000001	11	000000	Filp1	12
000002	06	000000	Filp1	27
000003	06	000000	Filp1	27
000004	06	000000	Filp1	118
000005	06	000000	Filp1	365
000006	06	000002	Filp1	365
000007	06	000003	Filp1	12

After running the triage script, each crash file in the output directory which passes through the script and hence crash data is printed. In the figure, target binary experienced a crash i.e. segmentation fault for each crash. This means memory corruption has occurred.

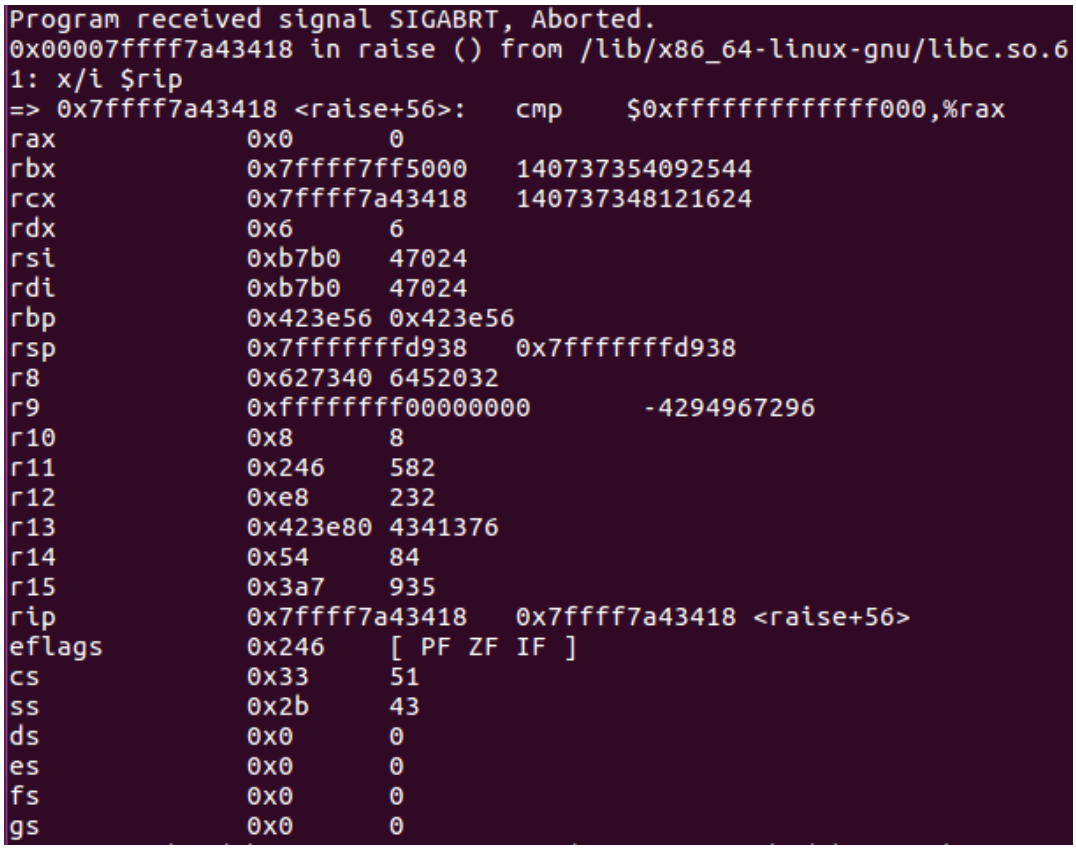

Figure 4.8 Memory Corruption

The exploitability of crashes and hangs may be ambiguous. Afl-fuzz tried to address by providing a crash exploration mode where a known-faulting test case is fuzzed in a manner similar to the normal operation of the fuzzer, but with a constraint that causes any non crashing mutations to be thrown away. The table below describes about the hangs that occurred.

Table 4.7 Hangs

Id	Source File	Operation	Position	Value
000000	000000	Filp1	290	-
000001	000000	Filp1	5	-
000002	000000	Filp1	92	-
000003	000000	Filp1	335	-
000004	000000	Arith8	195	-
000005	000000	Arith8	72	-9
000006	000002	Arith8	106	-18
000007	000003	Arith8	107	-22
000008	000003	Arith8	108	-3
000009	000003	Arith8	108	-9
000010	000003	Arith8	119	-14
000011	000003	Arith8	119	-25
000012	000003	Arith8	121	-29
000013	000003	Arith8	122	17
000014	000003	Arith8	124	-19
000015	000003	Arith8	132	13
000016	000003	Arith8	141	-6
000017	000003	Arith8	146	3

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

The SAFAL model proposed uses AFL which is a powerful fuzzer, leverages on source code and binaries which founded potential vulnerabilities. This is the first step in exploit development.

The model explored bugs in software and provided all the required metrics. It explored each executed path and on every path the possible vulnerability is founded. Stack based buffer overflow is the software vulnerability and the model proposed exploit this vulnerability in each executed path. Processing speed of fuzzer is fast and in short time it explored 72 paths in which 18 crashes and 8 hangs. For each crash bugs are found and at which memory address and what vulnerability occurs is determined by the model.

5.2 Future Scope

For Future work, the model can be applied to real-world softwares. Models in machine learning can be applied to this fuzzer to determine the efficiency of occurence of bugs and vulnerability, classify the vulnerabilities.

References

[1] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. "The oracel problem in software testing: A survey." *IEEE Transactions on Software Engineering* 41, no. 5 (2015): 507-525.

[2] Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., and Xie, T. "To be optimal or not in test-case prioritization." *IEEE Transactions on Software Engineering* 42, no. 5 (2016): 409-505.

[3]Huang, Bo, and Qiaoyan Wen. "An automatic fuzz testing method designed for detecting vulnerabilities on all protocol." *Computer Science and Network Technology (ICCSNT), International Conference, IEEE* 2 (2011).

[4]MDISS, Codenomicon. "Fuzz testing improving medical device quality and safety." *MIDSS TEchnical White Paper Series*, 2012.

[5] Lee, DoHoon, YoungHan Choi, and Jae-Cheol Ryou. "Api fuzz testing for security of libraries in windows systems: From faults to vulnerabilites."Convergence and Hybrid Information Technology." *ICCIT.Third International Conference on.IEEE* 2 (2008).

[6] Tsankov, Petar, Mohammad Torabi Dashti, and David Basin. "SECFUZZ: fuzz-testing security protocols." *Automation of Software Test (AST),7th International Workshop on. IEEE*, 2012.

[7] Avgerinos, Thanassis, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. "Automatic exploit generation." *Communications of the ACM* 2 (2014): 74-84.

[8] Mouzarani, Maryam, Babak Sadeghiyan, and Mohammad Zolfaghari. "Smart fuzzing method for detecting stack-based buffer overflow in binary codes." *IET Software* 10, no. 4 (2016): 96-107.

[9] Padaryan, Vartan A., V. V. Kaushan, and A. N. Fedotov. "Automated exploit generation for stack buffer overflow vulnerabilities." *Programming and Computer Software* 41, no. 6 (2015): 373-380.

[10] Meer, Deeb, Corelancoder and Wang. "Stack based buffer overflow exploitation." Tutorial.

[11] Tzermias, Zacharias. "Combining static and dynamic analysis for the detection of malicious documents." *Proceedings of the Fourth European Workshop on System Security. ACM*, 2011.

[12] Zhang, Ruoyu. "Combining static and dynamic analysis to discover software vulnerabilities." *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), Fifth International Conference on. IEEE*, 2011.

[13] Getman, Alexander, Vartan Padaryan, and Mikhail Solovyev. "Combined approach to solving problems in binary code analysis." *Proceedings of 9th International Conference on Computer Science and Information Technologies (CSIT)*, 2013.

[14] Caballero, Juan. "Input generation via decomposition and re-stitching: Finding bugs in malware." *Proceedings of the 17th ACM conference on Computer and communications security. ACM*, 2010.

[15] Avgerinos, Thanassis. "Automatic exploit generation." Communications of the ACM." *Communications of the ACM* 57, no. 2 (2014): 74-84.

[16] Wang, Xinran. "Sigfree: A signature-free buffer overflow attack blocker." *IEEE transactions on dependable and secure computing* 7, no. 1 (2010): 65-79.

[17] Francillon, Aurélien, and Claude Castelluccia. "Code injection attacks on harvard-architecture devices." *Proceedings of the 15th ACM conference on Computer and communications security. ACM*, 2008.

[18] Tian, Donghai. "Defeating buffer overflow attacks via virtualization." *Computers & Electrical Engineering* 40, no. 6 (2014): 1940-1950.

[19] Wilander, John, and Mariam Kamkar. "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention." *NDSS* 3 (2003).

[20] Petroni Jr, Nick L., and Michael Hicks. "Automated detection of persistent kernel control-flow attacks." *Proceedings of the 14th ACM conference on Computer and communications security. ACM*, 2007.

[21] Szekeres, L., Payer, M., Wei, T. "Sok: Eternal war in memory." *IEEE Symp. on Security and Privacy (SP)*, 2013: 48-62.

[22] Cui, Baojiang. "WhirlingFuzzwork: a taint-analysis-based API in-memory fuzzing framework." *Soft Computing* 12, no. 21 (2016): 3401-3414.

[23] Mouzarani, Maryam, Babak Sadeghiyan, and Mohammad Zolfaghari. "A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes." *21st Pacific Rim International Symposium on. IEEE*, 2015.

[24] Chen, Kai, Yingjun Zhang, and Peng Liu. "Dynamically discovering likely memory layout to perform accurate fuzzing." *IEEE Transactions on Reliability* 65, no. 3 (2016): 1180-1194.

[25] Kim, Joon-Ho, Myung-Chul Ma, and Jae-Pyo Park. "An analysis on secure coding using symbolic execution engine." *Journal of Computer Virology and Hacking Techniques* 3, no. 12 (2016): 177-184.

[26] Chen, Li-Han. "A Robust Kernel-Based Solution to Control-Hijacking Buffer Overflow Attacks." *J. Inf. Sci. Eng* 27, no. 3 (2011): 869-890.

[27] Wang, Yong,. "RICB: Integer overflow vulnerability dynamic analysis via buffer overflow." *International Conference on Forensics in Telecommunications, Information, and Multimedia. Springer*, 2010.

[28] Sidiroglou, Stelios, Giannis Giovanidis, and Angelos D. Keromytis. "A dynamic mechanism for recovering from buffer overflow attacks." *ISC*, 2005.

[29] Berry, Chris. "Hunting For Bugs With AFL 101 - A PRIMER." Aura Information Security Research Blog. January 23, 2017. Accessed July 09, 2017.

[30] DeMott, Jared. "The evolving art of fuzzing." (DEF CON) 14 (2016).

[31] Oehlert, Peter. "Violating Assumptions with Fuzzing." *IEEE Security & Privacy*, March/April 2005: 58-62.

[32] Chen, Yang. "Taming compiler fuzzers." *ACM SIGPLAN Notices* 48, no. 6 (2013).

[33] Banks, Greg. "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr." *ISC* 4176 (2006).

[34] Rawat, Sanjay. "Vuzzer: Application-aware evolutionary fuzzing." *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[35] Nagaraja, Leeka, "Evaluation of File Format Fuzzing Tool: MiniFuzz." *Network Security,2011*

List of Publications

[1] Manisha Bhardwaj and Seema Bawa, ***"Stack Based Buffer Overflow: A Survey"***, The International Conference on Recent Advancement in Computer, Communication and Computational Sciences (RACCCS-2017).
[Communicated]

[2] Manisha Bhardwaj and Seema Bawa, ***"Fuzz Testing in Stack Based Buffer Overflow"***, IC4S 2017: 2nd International Conference on Computer, Communication and Computational Sciences.
[Communicated]