DESERVE: A FRAMEWORK FOR DETECTING PROGRAM SECURITY VULNERABILITY EXPLOITATIONS

Amatul Mohosina

A thesis submitted to the Department of Electrical and Computer Engineering

in conformity with the requirements for

the degree of Masters of Applied Science

Queen's University

Kingston, Ontario, Canada

(September, 2011)

Abstract

It is difficult to develop a program that is completely free from vulnerabilities. Despite the applications of many approaches to secure programs, vulnerability exploitations occur in real world in large numbers. Exploitations of vulnerabilities may corrupt memory spaces and program states, lead to denial of services and authorization bypassing, provide attackers the access to authorization information, and leak sensitive information. Monitoring at the program code level can be a way of vulnerability exploitation detection at runtime. In this work, we propose a monitor embedding framework DESERVE (a framework for DEtecting program SEcuRity Vulnerability Exploitations). DESERVE identifies exploitable statements from source code based on static backward slicing and embeds necessary code to detect attacks. During the deployment stage, the enhanced programs execute exploitable statements in a separate test environment. Unlike traditional monitors that extract and store program state information to compare with vulnerable free program states to detect exploitation, our approach does not need to save state information. Moreover, the slicing technique allows us to avoid the tracking of fine grained level of information about runtime program environments such as input flow and memory state. We implement DESERVE for detecting buffer overflow, SQL injection, and cross-site scripting attacks. We evaluate our approach for real world programs implemented in C and PHP languages. The results show that the approach can detect some of the well-known attacks. Moreover, the approach imposes negligible runtime overhead.

Acknowledgement

I am grateful and indebted to my supervisor Dr. Mohammad Zulkernine, School of Computing, Queen's University, for his excellent support, advice, and expertise. He always guided me in the right direction. Without his contribution, this work would not have been possible.

I am very thankful to my co-worker Hossain Shahriar at Queen's Reliable Software Technology (QRST) lab for helping me revise this thesis. I would like to thank the members of the QRST group, who supported me through many enjoyable discussions.

I would also like to thank my parents and siblings for their support and encouragement. A special thanks to my husband Subir, to be right beside me always to break my procrastination.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In all aspects of human life, software is now widely used to store, process, and transfer sensitive and confidential information. Unfortunately, implemented software contains vulnerabilities at program code level. These vulnerabilities can be exploited to gain access to sensitive information and execute arbitrary code. The number of vulnerability exploitations due to program code level vulnerabilities has increased over the last few years [1]. The reported financial and non-financial losses incurred due to these exploitations have been found huge [62]. Related stake holders often lose confidence on a deployed program due to program vulnerabilities. Vulnerabilities in real-time software (software related to nuclear reactor, airplane etc.) may pose serious risk to human life. Thus, the mitigation of vulnerability exploitations is extremely important.

Software testing and monitoring are two widely used vulnerability detection techniques. Software testing identifies vulnerabilities before deployment. Software monitoring detects vulnerability exploitations at runtime. The vulnerability detection techniques at both phases mainly apply static [2-6] and dynamic [7-11] analysis. Static analysis investigates the source code without executing the program; on the other hand, dynamic analysis executes a program, collects a trace, and analyzes it. Static analysis

aims to predict all weak points of a source code. It sometimes may identify safe statements as weak and generate many false positives. Thus, static analysis is complete but not sound. Alternatively, dynamic analysis executes a program with actual inputs. Dynamic analysis can detect a vulnerability correctly. This property makes the analysis technique sound. However, it needs to execute a program with all possible inputs to detect all vulnerabilities. Executing a program with all possible inputs is not feasible [12, 13]. Thus, dynamic analysis is not complete.

1.2 Motivation

Even after extensive testing, some vulnerabilities may remain uncovered. Software monitoring can be a way to prevent the exploitation of those undetected vulnerabilities through early detection and prevention.

Traditional monitors examine certain properties of a program to detect attacks denoted as *monitoring aspect* [14]. These properties normally either create an attack or are altered after an attack. A monitoring aspect for a buffer overflow monitor can be a buffer modifying function call, functions' return addresses, or an execution flow of a program. An SQL injection monitor can examine database access functions and an XSS monitor can check HTML generation functions to detect attacks. A vulnerability monitor examines the changes in a monitoring aspect. Whenever there is a change in an instance of the monitoring aspect, the monitor extracts, analyzes, saves or compares program state information to detect vulnerability exploitations. Figure 1.1 shows the working principle of a typical monitor. This monitor collects information from certain points of a program

execution; processes the information; and finally compares this information with signatures located in a database. Some monitors store the extracted information into the signature and other database. If a program's actual behavior is deviated from the expected behavior, the monitor perceives it as an exploitation.

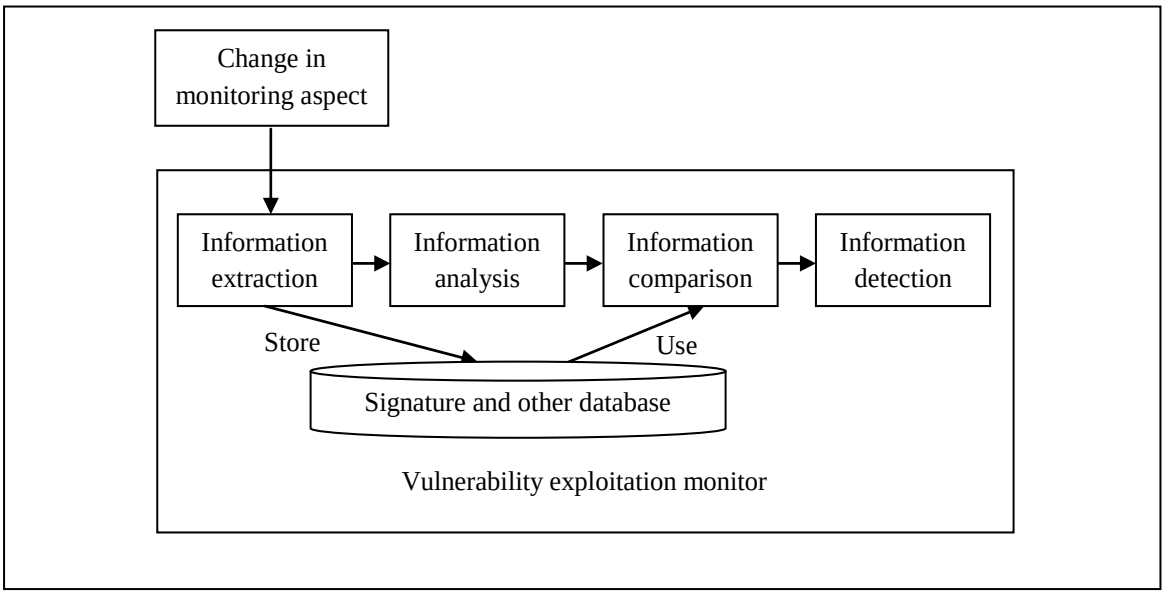

Figure 1.1: A traditional vulnerability monitor

Monitors keep track of certain changes of monitoring aspects. Whenever there is a change in the monitoring aspect value, normally a program state of the running program is compared with a known program state to detect an attack. The traditional monitoring approaches have the following limitations:

First, traditional monitors keep track of every change of monitoring aspects' value [14]. A change in monitoring aspects' value does not always introduce an exploitation. Some monitoring techniques apply dynamic taint analysis to track only exploitable ones. Dynamic taint analysis marks a user input and tracks it at run time. It is

also known as the dynamic information flow analysis. Dynamic taint analysis may introduce runtime overhead [7, 20]. Second, attackers provide new inputs to exploit vulnerabilities. If the program state checking occurs before an exploitation, the crafted inputs of the attackers may lead to the states that may appear safe, yet create attacks. If the program state checking occurs after the exploitation, the checking does not help.

1.3 Contribution

We propose a vulnerability monitor embedding framework DESERVE (a framework for DEtecting program SEcuRity Vulnerability Exploitations) which identifies potential vulnerable statements in an application source code by applying static backward slicing. DESERVE then embeds monitoring code for those statements. The embedded monitor executes and tests a potential vulnerable statement in a separate test environment. If the result of the test does not indicate any exploitation, the original process resumes the execution.

The specific contributions of this thesis are as follows:

1. The monitors embedded by our proposed framework DESERVE detects three widely known vulnerabilities without extracting program state information that include Buffer Overflow (BOF), SQL Injection (SQLI), and Cross Site Scripting (XSS). These monitors execute and test a vulnerable statement in a separate test environment before executing it in the original process. We provide general guidelines to design the runtime test environment. We define necessary properties of the runtime test environments for BOF, SQLI, and XSS

monitoring. We also implement the framework for BOF, SQLI, and XSS monitoring. We apply DESERVE to embed monitors in real world C and PHP programs. The embedded monitors can successfully detect the exploitations.

2. To achieve a reasonably low performance overhead, the embedded monitors do not initiate a test process for all *vulnerable statements*. By vulnerable statements, we refer to those statements whose execution can cause vulnerability exploitations. DESERVE applies static backward slicing to find out *exploitable vulnerable statements* and instrument source code only at those vulnerable statements. By exploitable vulnerable statement, we mean those vulnerable statements whose executions are being influenced by any user input. We evaluate the performance of DESERVE by applying it on real world programs. For these programs, DESERVE can detect many well-known BOF, SQLI, and XSS vulnerabilities.

1.4 Thesis organization

The organization of this thesis is as follows. In Chapter 2, we provide necessary background on three vulnerabilities (BOF, SQLI, and XSS). We also discuss static backward slicing and runtime time testing followed by the related works on these vulnerabilities. We discuss our proposed monitor embedding framework DESERVE in Chapter 3. Chapters 4-6 provide details of the monitor embedding to detect BOF, SQLI, and XSS attacks, respectively. Finally, we conclude the thesis and discuss the limitations and future work in Chapter 7.

Chapter 2

Background and Related Work

In this chapter, we provide the background knowledge which is essential to understand this thesis. In addition to the background knowledge, we also summarize the works related to our research. In Section 2.1, we provide short descriptions of the vulnerabilities we address in this thesis. We also explain the detection techniques which are related to our work in Section 2.2. We present related research works on buffer overflow, SQL injection, and cross-site scripting monitoring in Section 2.3. We also discuss this thesis work in the context of these research works. Section 2.4 summarizes the chapter.

2.1 Vulnerabilities

Despite many initiatives, buffer overflow, SQL injection, and cross-site scripting are still among the top twenty five most dangerous software security errors [1]. We implement DESERVE to embed monitors for these three vulnerabilities. In the following subsections, we briefly discuss about these vulnerabilities.

Among the above-mentioned vulnerabilities, SQL injection and cross-site scripting are well known code injection attacks. Code injection is a vulnerability exploitation technique. A malicious user injects code (SQL, javascript) to change the course of the execution of a program. Depending on the time of action, code injection

techniques can be divided into two categories: first and second order attack. The injected code in the first order attack is executed immediately. In the second order attack, the injected code is stored in a database. When the data is fetched from the database, the attack occurs [15, 16]. The second order attack can be more devastating. Our proposed framework DESERVE can embed monitors for both first and second order attacks. In this section, we discuss both first and second order code injection attacks with examples.

2.1.1 Buffer overflow

A BOF is an implementation flaw that results in exceeding the capacity of a destination buffer while copying contents from a source buffer. The overflow modifies the contents of the adjacent memory locations of a destination buffer. As a result, a program state might get corrupted and the execution flow of a program can be altered by overwriting sensitive memory locations such as a return address or a stack frame pointer.

If a memory writing operation overwrites any function return address; after the end of a function execution, a program might continue executing from an unexpected memory location. To exploit a BOF vulnerability, a malicious user pre-fills that memory location with his or her desired command. We present an example of BOF in Figure 2.1.

Figure 2.1(a) shows a function *foo* that can cause a BOF. *t* is a character array of length 2. The execution of *strcpy* function in this code segment overflows the adjacent memory spaces. Figure 2.1(b) shows the status of the stack before *strcpy* function call. The first two positions are reserved for the array *t*. After these cells, there is an empty cell. This cell is followed by a memory cell having the return address of this function:

reAd. The execution of *strcpy* function overwrites the *reAd* with a value 4. At the end of the function execution, the program resumes the execution from the command at memory location 4 which may not even in the program memory space. Table 4.1 in Chapter 4 presents a list of C library functions which can introduce vulnerabilities in a program. These functions mainly copy /merge string contents, provide formatted output to a string, or take input into strings. Besides these function-calls, improper access to buffers through array indexes and pointers can also cause buffer overflow.

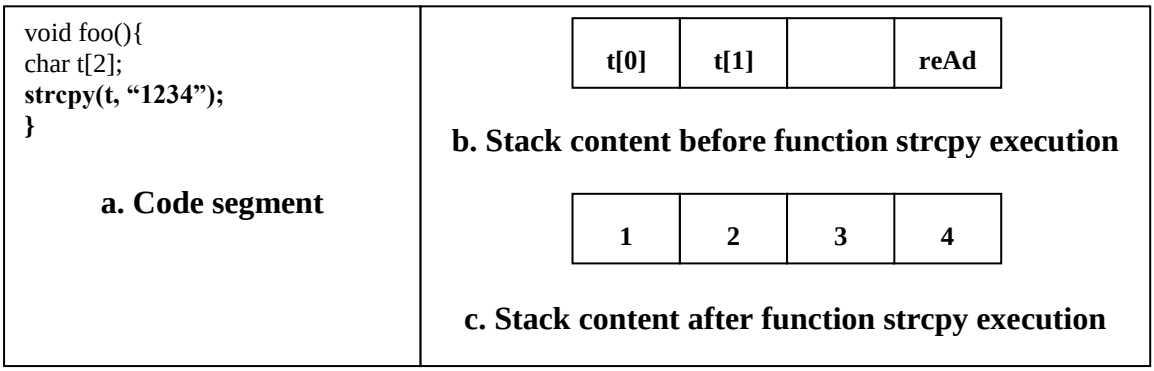

Figure 2.1: An example of BOF

2.1.2 SQL injection

SQLI is a well-known code injection attack. To exploit an SQLI vulnerability, instead of a string, a malicious user inserts an SQL code into an input field of a web application. Database queries are generated through server side scripts written in PHP, C#, and Java. A web application employs a database to store and retrieve information. Later, the inserted code is executed in the database server. This execution can cause an unexpected program behavior.

To explain an SQLI attack, we introduce a login example. A user can login into a web application using user name *amatul* and password *4test*. The server script sends the following query to the database server: *Select * from User where uid='amatul' and pwd='4test*'. If the user exists, the query returns one or more rows. The script checks the number of rows in result. If the number is greater than zero, it allows the user to log in.

Now let us consider an attack scenario. A malicious user inserts *'or 1=1;*-- as the username, and *blahblah* as the password. Therefore, the SQL query sent to the database server has the following structure: *Select * from User where uid='' or 1=1 --' and pwd='blahblah'.* -- indicates the beginning of a comment in SQL. As a result; the database server omits the remaining portion of the SQL query after --. It executes the following query: *Select * from User where uid='' or 1=1*. *uid='' or 1=1* is a condition which is always true. After executing the query, the database server returns the full dataset of *User* data table and a user can log in without any real username or password.

SQLI can be divided into first and second order attack. A first order attack takes place immediately after inserting a malicious SQL script. In a first order attack, the SQL script is not saved, so the attacker inserted script is not reusable. Attackers need to insert another SQL script again to introduce a new attack. The previous scenario is an example of a first order attack.

A second order attack does not occur right away. The malicious script is saved into a database instead. Later, during the data fetching, an attack occurs. Let us consider a typical credit card list page of an e-commerce website for user *Brie*. Normally users

save their credit card information into an e-commerce website for payment convenience. For a user *Brie,* the list of the credit card is generated from the result of a query: *select * from credit_card_info where user_name='Brie'*. Here, table *credit_card_info* stores credit card information for each user. A registered user can save, modify, delete, and see his or her credit card information through the credit card list page. A malicious user who has an intention to do a second order attack, can fill the registration form of the website with malicious user name *' or 1=1* --. Now whenever this malicious user accesses to the credit card list page, the list generation query takes the following structure: *select * from credit_card_info where user_name='' or 1=1* --. As *1=1* is always true, the execution of the query returns all data rows from *credit_card_info* table. Thus, the malicious user can see credit card information of all users and can modify credit card information. As the script has been saved into the database, the attacker can repeat this attack without injecting any new script. Our proposed framework DESERVE can effectively embed a monitor for both first and second order attacks of SQLI.

Depending on the inserted SQL script and the attacker intention, SQLI can be classified into several types. Halfond et al [15] describe different types of SQLI. Table 2.1 shows the classification, example and attacker intention for each type of attack. To detect an attack, the SQLI monitor embedded by DESERVE does not need to identify an attack type. It tests all types of SQLI attacks in the same manner.

Table 2.1: SQLI attack type [15]

SQLI type	Inserted script example	Attacker intention
Tautologies	’ or 1=1 --	1. Bypassing authentication 2. Identifying injectable parameters 3. Extracting data
Union query	’ UNION SELECT cardNo from CreditCards where acctNo=10032 --	1. Bypassing authentication 2. Extracting data
Piggy backed queries	pass=’’; drop table users --	1. Extracting data 2. Adding or modifying data 3. Performing denial of service 4. Executing remote commands
Inference	Blind injection	1. Identifying injectable parameters 2. Extracting data 3. Determining database schema
Alternate Encodings	’; exec(0x73687574646f776e) -- (SHUTDOWN.”)	Evading detection
Illegal/Logically Incorrect Queries	“convert(int,(select top 1 name from sysobjects where xtype=’u’))”	1. Identifying injectable parameters 2. Performing database finger-printing 3. Extracting data
Stored Procedures	Any of the above	1. Performing privilege escalation 2. Performing denial of service 3. Executing remote commands

2.1.3 Cross-site scripting

A cross-site scripting (XSS) attack occurs when a malicious user inserts client side scripts into the input field of a web page. Later, when another user views such a page, the code is executed by the web browser and an attack occurs.

In a typical XSS attack, the malicious script is sent to the server as an HTTP GET or POST request, and an attack occurs immediately after the execution of the response from the server. It is also called reflected or first order attack. Google code University [16] provides a web application *Gruyere* to perform and understand malicious attacks. URL *http://google-gruyere.appspot.com/367839749691/snippets.gtl?uid=brie* of

Gruyere shows the list of snippets posted by user *brie*. The *uid* is sent to the server using http GET request. In response to this request, the server sends back a snippets list page by showing the *uid* at the top of that page. To exploit an XSS vulnerability, a malicious user replaces the *uid brie* with a JavaScript code *<script>alert(1)</script>*. During the server response interpretation, a browser finds the *uid* as a JavaScript code and executes it. This execution causes the popping up a new window. By social engineering, a malicious user convinces a normal user to click this modified URL. In this example, the inserted script is not dangerous. However, a carefully inserted script can collect and modify sensitive user information.

XSS is a code injection attack. XSS attacks can be divided into first and second order attack. The previous case is an example of the first order XSS attack. We present an example of the second order attack for the same application *Guyere*. A registered user of this web application can create a new snippet. The snippet is saved to a database. A user can see the snippets created by the other users. To show all snippets, the server site script fetches all snippets from the database and put the information into the website HTML code. A malicious user inserts code *read this!* into new snippet text area. The JavaScript is saved as snippet content and appears in the snippet list. On placing the mouse pointer over this snippet, the inserted code pops up an alert window. In this example, the inserted script is not dangerous. However, a carefully inserted script can collect and modify sensitive user information. This scenario is an example of a second order or stored attack. The script provided by malicious user is saved into a database and whenever that database content is fetched and

shown in a webpage, an attack occurs. A stored attack is more devastating than a first order attack, as a normal user is affected even without clicking any malicious URL. Our proposed framework DESERVE can effectively embed monitor for both first and second order attacks.

Depending on who is processing the HTML content with malicious script, the XSS attack can be divided into traditional server side scripting vulnerability and DOM (Document Object Model) based vulnerability. Traditionally, a server side script generates HTML contents with malicious code and XSS attacks occur. DOM-based vulnerabilities occur when HTML contents is processed by Javascript. In our work, we consider server-side scripting XSS vulnerability only.

2.2 Detection techniques

In this section, we discuss about static backward slicing and runtime testing. Static backward slicing is a well known approach for source code analysis. We use this approach to identify the vulnerable statements which can be influenced by user inputs. We use runtime testing to monitor vulnerability exploitation. We describe an existing runtime testing approach: *in vivo* testing. *In vivo* testing is a kind of runtime testing which logs test result for the future use. As we use runtime testing, we also compare *in vivo* testing with the runtime testing used in this thesis.

2.2.1 Static backward slicing

Backward slicing of a source code at a program point is a subset of statements from the source code, which have influence on that program point [17]. There is an

example code snippet in Figure 2.2. The behavior of the execution of Line 7 depends on the value of variable *arr2*. The statements in Lines 6, 4, 2, and 1 influence the value of *arr2* directly or indirectly. To determine a backward slice, we need to define slicing criterion, *C= (p, V)*, where *p* is the statement whose slice we are going to find and *V* is a subset of the program variables. Initially, *V* has the variables used in *p*. To determine the slice, the slicing algorithm walks backward through program statements. If a statement modifies any variable *v* in *V*, it is added to the slice. The variables in the modifying expression are being added to *V*. These steps are being repeated up to the starting of a function. For the program in Figure 2.2, *p* is the statement at Line 7. *V* initially contains only *arr2*. However, *arr2* is updated by the statement at Line 6. Line 6 is put in the slice and variable *b* is added to *V*. Line 4 modifies variable *b*. It is also put in the slice. Lines 1 and 2 contain the variables *b* and *arr2* declarations, respectively. These are also placed in the slice. Finally, the slice contains Lines 1, 2, 4, and 6. The statements in the slice are shown in grey background in Figure 2.2.

```
void main() {
1. int a,b;
2. char arr1[20], arr2[20];
3.a=18;
4. b=10;
5. arr1[a]='c';
6. arr2[b]='d';
7. printf("%s", arr2);
}
```

Figure 2.2: An example of static backward program slicing

2.2.2 Runtime testing

Runtime testing is a unit testing which is performed during deployment. The test is initiated at a program point of a running program and performed in an isolated test environment. *In vivo* testing is an example of runtime testing [18]. As the application executes, tests also execute continuously at arbitrary program points. The unit testing is done in a sandboxed environment and does not affect the original course of the program. Test results are logged for future use. Source code instrumentation is needed to apply *in vivo* testing in an application.

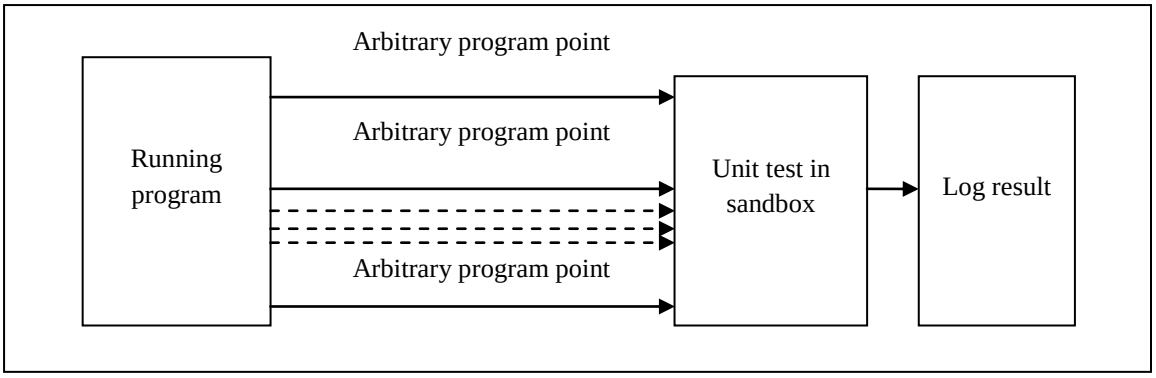

Figure 2.3: *In vivo* testing

Figure 2.3 shows an overview of *in vivo* testing. The leftmost box represents a running application. The arrows denote arbitrary program points of the running application. A unit testing takes place at an arbitrary point in a sandboxed environment. The box in the middle represents the unit testing in sandboxed environment. Finally, the box at the right denotes the logging of the test result.

Murphy *et al* [18] propose *in vivo* testing for bug detection. Later, they apply this testing approach along with fuzz testing to detect software vulnerability [19]. *In vivo*

testing starts at arbitrary program points in a sandboxed environment and executes without hampering the course of the original program. A monitor embedded by DESERVE also initiates a runtime testing. However, this monitor starts testing vulnerable statements at predefined program points (determine by static slicing) in a separate test environment. The original program waits for a safety response from the test environment. The basic differences between *in vivo* testing and runtime testing used in this thesis are listed below:

1. *In vivo* testing initiates a test at random program points. On the other hand, a monitor embedded by DESERVE initiates a test only for exploitable vulnerable statements.
2. The behavior of the original program is not influenced by the test outcome in *in vivo* testing. Alternatively, in our monitoring approach, the original program waits for the test completion and uses the runtime test outcome to recognize an exploitation.
3. The test result of *in vivo* testing is stored and used in future for test case generation and program analysis. However, we use the results of runtime testing to monitor an attack.

2.3 Related work

In this section, we discuss the research works which cover BOF, SQLI, and XSS monitoring. We also briefly present some related security frameworks specially for monitoring the above-mentioned vulnerabilities. Lam *et al* [9] propose a generalized

framework for security monitoring. They propose a dynamic taint tracing framework to track the information flow of C programs for security. To reduce the information flow tracking overhead, they apply static backward and forward slicing. They also develop a tool called “Aussum” which does selective sandboxing of network application execution if the inputs are tainted. The monitor embedded by our framework DESERVE does not need dynamic taint tracking. DESERVE applies only backward slicing to recognize exploitable statements. The monitors embedded by DESERVE do not execute the original program execution in a sandbox; rather those monitors test the execution of the exploitable statement in a separate test environment. If the exploitable statement passes the test, the original program also executes the exploitable statement with full privilege. So far DESERVE is implemented for BOF, SQLI, and XSS monitoring of C and PHP applications. It is extendable for new vulnerabilities of other programming languages.

Current BOF, SQLI, and XSS monitoring works are discussed in the following subsections. The frameworks for BOF, SQLI, and XSS monitoring are also summarized under the respective subsections. We present a comparative picture of this thesis in the context of those works. We discuss BOF vulnerability monitoring works in Subsection 2.1. In Subsection 2.3.2, we describe SQLI monitoring works. Finally, XSS monitoring research works are discussed in Subsection 2.3.3.

2.3.1 Buffer overflow

Del Grosso *et al* [3] apply genetic algorithm to generate test cases for BOF vulnerability detection. To reduce the input domain, they apply static slicing. The attack

detection success of this work depends on the generation of adequate test cases. DESERVE also applies static backward slicing. However, DESERVE uses it to determine if user inputs have influence on a vulnerable statement.

Clause *et al* [7] mark all return values from any input taking function as tainted. Dalton *et al* [20] also have a research work where they employ dynamic taint analysis. In these works, the tainted flow of a program is determined during runtime. If the tainted variables are used in any vulnerable function or as a jump address, an attack may occur. They assume that any input from a user can be dangerous if that is used directly or indirectly by any library function or as a function return address. They use dynamic taint flow tracking to determine tainted variables. Dynamic taint flow tracking can determine a tainted variable accurately. However, the tracking of the tainted flow creates huge runtime overhead. We also assume that input from a user can be dangerous, if it is used in any vulnerable statement. Instead of dynamic taint analysis, DESERVE analyzes the source code by applying static backward slicing to identify if a user input is used by any buffer modifying statement. This procedure may generate some false positives. Therefore, our framework embeds monitors to test the buffer modifying statements at runtime. Some other works apply dynamic taint tracking to detect BOF [10, 21]. A monitor embedded by DESERVE does not need dynamic taint tracking to detect an attack.

Many BOF monitoring works save the return address of a function from any corruption by buffer overrun [8, 22-27]. They extract the return address from the flow of a program and save it. Later, they compare the current return address to the saved return

address to detect an attack. Normally, these monitors are implemented using code instrumentation or binary rewriting. This procedure needs extra storage to save the return addresses. The monitor embedded by DESERVE does not need to know a function return address to detect an attack.

Some works apply boundary checking to detect a BOF attack [28-31]. Some research works keep track of memory allocations [11, 32, 33]. Akritidis *et al* [34] apply both boundary checking and memory allocation tracking to detect BOF. Boundary checking for each buffer modifying statement is costly, and keeping track of memory allocation introduces storage and time overheads. The BOF monitor embedded by DESERVE does not check the boundary to detect a BOF attack. It tests a vulnerable statement in a test process and checks if it corrupts the buffer. It also does not need to keep track of memory allocation.

In some research, the running behavior of a program is compared with known standard behavior of the program under monitoring [35-37]. The BOF monitor embedded by DESERVE does not need to maintain the standard program behavior to detect an attack. Memory randomization/rearrangement/protection and machine code randomization are also applied to detect BOF [38-44].

2.3.2 SQL injection

Clause *et al* [7] use positive tainting to detect an SQLI. They identify all trusted strings and make sure that sensitive portions of an SQL query use only the trusted strings. Dalton *et al* [20] also have a research work where they employ dynamic taint analysis. In

this research, the tainted flow of a program is determined during runtime. If the tainted variables are used in any vulnerable function, an attack may occur. They assume that any input from a user can be dangerous, if that is used directly or indirectly by any vulnerable function. They use dynamic taint flow tracking to determine tainted variables. Dynamic taint flow tracking can determine a tainted variable precisely. However, taint tracking creates heavyweight runtime overhead. We also assume that the influence of user inputs on a database access function can cause an SQLI exploitation. However, our framework DESERVE analyzes the source code using static backward slicing to find out if any user input is used by any vulnerable function. This procedure may lead to some false positives. To overcome this limitation, our framework embeds a monitor to test the buffer modifying statement at runtime.

Bandhakavi *et al* [45] compare the runtime parse tree with an intended parse tree of SQL query. The intended parse tree is extracted from source code. This approach needs byte-code transformation which is costly. Buehrer *et al* [46] also compare the standard parse tree of SQL and actual query to detect SQLI. However, our SQLI monitor does not need to build the parse tree to detect an attack.

Alfantookh *et al* [47] intercept HTTP requests and remove the suspicious characters to detect an SQLI attack. The success of this approach depends on the completeness of the suspicious character list. This approach may not be effective to detect a zero-day attack. In contrast, the SQLI monitor by DESERVE does not need to maintain any suspicious character list to detect an attack.

Boyd *et al* [48] apply a randomization of SQL queries using a key at client-site proxy. Later, the web server de-randomizes these. One of the main problems of this approach is that the set up is complex. Also, if the key is exposed, this approach cannot prevent SQLI attacks. As client side proxy has the key of randomization, it is possible that the key can be exposed easily.

2.3.3 Cross-site scripting

Dalton *et al* [20] have a research work where they employ dynamic taint analysis to determine the tainted flow of a program during runtime. If the tainted variables are used in any vulnerable function, an attack may occur. Dynamic taint flow tracking can determine a tainted variable accurately. However, following the taint flow creates heavyweight runtime overhead. Although, we have the same intention of indentifying the vulnerable statement influenced by the user input, DESERVE analyzes the source code using static program slicing to identify if any user input is used by a vulnerable function. This procedure may lead to some false positives. Therefore, our framework embeds a monitor to test the buffer modifying statement at runtime.

Bhatia *et al* [71] employ slicing to find weak points of Java web programs and suggest the fixes at development time. Our framework employs static backward slicing on PHP programs after development to find weak points and embeds monitor at those points.

Some XSS detection works compare the JavaScript code in a HTTP response to a shadow page/known list of scripts [49, 50]. In contrast, the XSS monitor embedded by

DESERVE may not require a comparison for every HTTP response. As a result, the overhead is not introduced for each HTTP response. Wurzinger *et al* [51] replace the JavaScript code with a unique ID and let malicious script execute to identify an attack. We do not alter JavaScript code to detect an attack. Kirda *et al* [52] apply customized firewall rules to allow or disallow HTTP requests and responses to prevent an XSS attack. DESERVE does not need to customize firewall rules. Jim *et al* [53] insert security policies in legitimate JavaScript code from the server so that the browser can interpret those to prevent XSS attacks. The work of Ismail *et al* [54] replaces HTML special characters in an HTTP request and response using an encoding mechanism. Our monitor does not need to consider HTML special characters. Iha *et al* [55] employ binding method to detect XSS attacks.

2.4 Summary

In this chapter, we have provided the literature review and related background materials. We have provided brief descriptions and examples of BOF, SQLI, and XSS exploitations. We have also explained the detection techniques related to our work. We have presented the summary of works which use similar techniques either to detect an exploitation or to monitor any of the vulnerabilities we consider in this thesis. We have compared and contrasted our works with the current works in BOF, SQLI, and XSS monitoring. In the next chapter, we will present the details of the proposed monitor embedding framework.

Chapter 3

The Monitor Embedding Framework

In this chapter, we discuss the monitor embedding process in details. The proposed monitor embedding framework DESERVE processes program source code in three consecutive steps. We explain the working principle of the embedded monitors. We discuss about the attributes, properties, inputs, and outputs of the framework. This chapter also provides necessary and sufficient guidelines to implement the framework for a new vulnerability. We present the detail of the monitor embedding framework in Section 3.1 and describe the runtime behavior of the monitor in Section 3.2.

3.1 Steps of the framework DESERVE

Figure 3.1 shows a high level view of the framework DESERVE. DESERVE takes the source code as input and delivers the instrumented source code as output. The instrumented program is capable of monitoring itself after the deployment. The left box denotes the framework DESERVE and the right box indicates the deployment of the instrumented source code. DESERVE embeds the monitor into source code at exploitable vulnerable statements. Figure 3.2 shows the three consecutive steps of DESERVE. The rectangular boxes represent the steps of DESERVE. DESERVE processes the application source code based on the consecutive steps: vulnerable statement marking, exploitable vulnerable statement marking, and code instrumentation. The thin arrows denote inputs

and outputs. A thick arrow between two boxes represents that the output of the source box serves as input to the destination box.

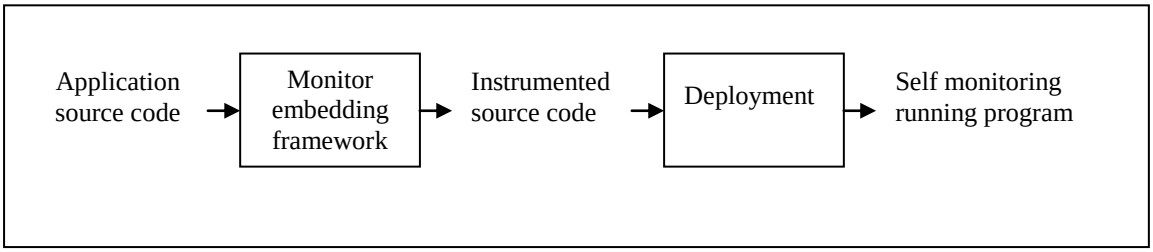

Figure 3.1: Working principle of the framework DESERVE

Subsections 3.1.1-3.1.3 describe these three steps in detail. In Subsection 3.1.1, we define and describe the statement patterns and function-calls that we consider as *vulnerable statement*. It also describes the way of automatically marking those statements. The outcome of this step serves as input to the next step. In the second step, DESERVE applies backward slicing to determine *exploitable vulnerable statements* among all vulnerable statements. In Subsection 3.1.3, we describe the process of source code instrumentation. We discuss about the implementation of monitors for different vulnerabilities. We also describe necessary attributes to design a *runtime testing environment* and provide the guidelines to conduct a runtime testing.

To implement all the three steps of DESERVE, we develop three TXL programs [56]. TXL is a programming language which is used for source transformation and analysis. It is a rule/function based language which transforms the source according to the rules/functions. To transform the source code, a TXL program needs two components: grammar of application language written in TXL and transformation

rules/functions. Figure 3.3 shows the working principle of TXL rules/functions. It needs transformation rules and the language grammar to apply the transformation on source code. A TXL program needs the grammar of application language to determine the parse tree and applies the rules/functions to modify a specific node. To run the steps of the framework sequentially, we develop a Python script [64]. As a result, DESERVE can instrument the application source code automatically.

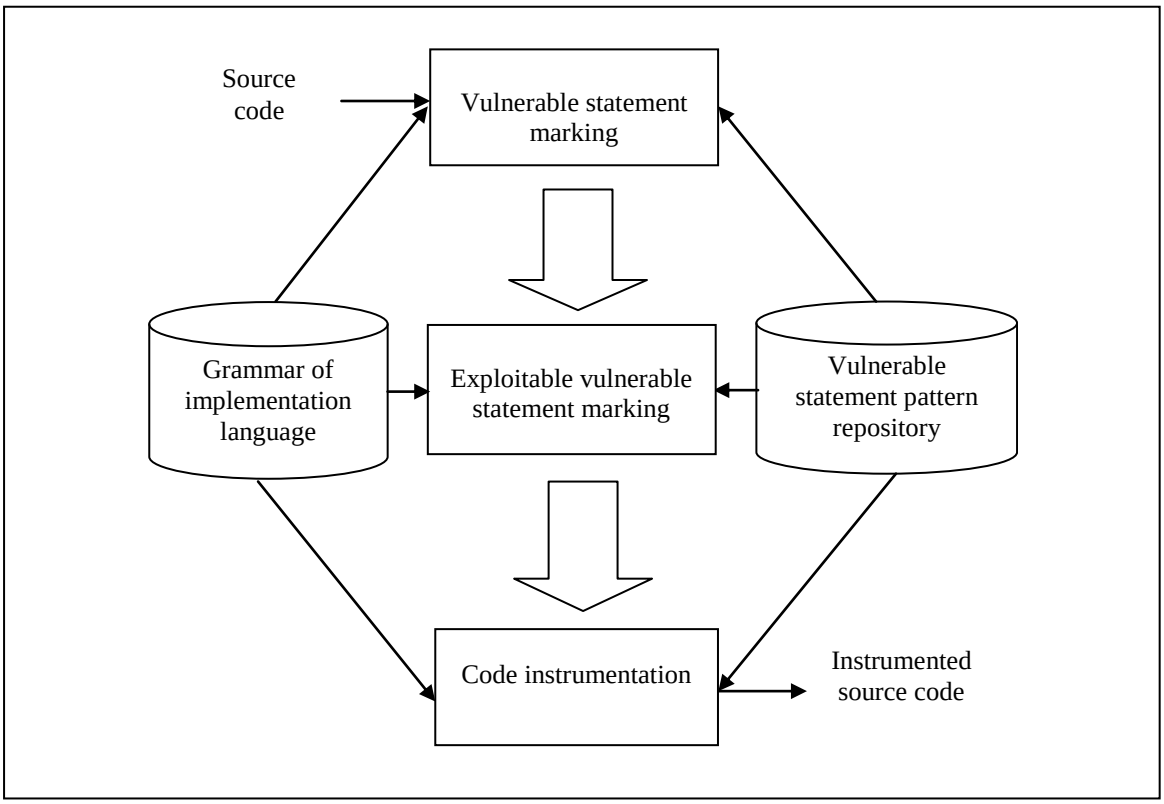

Figure 3.2: The monitor embedding framework

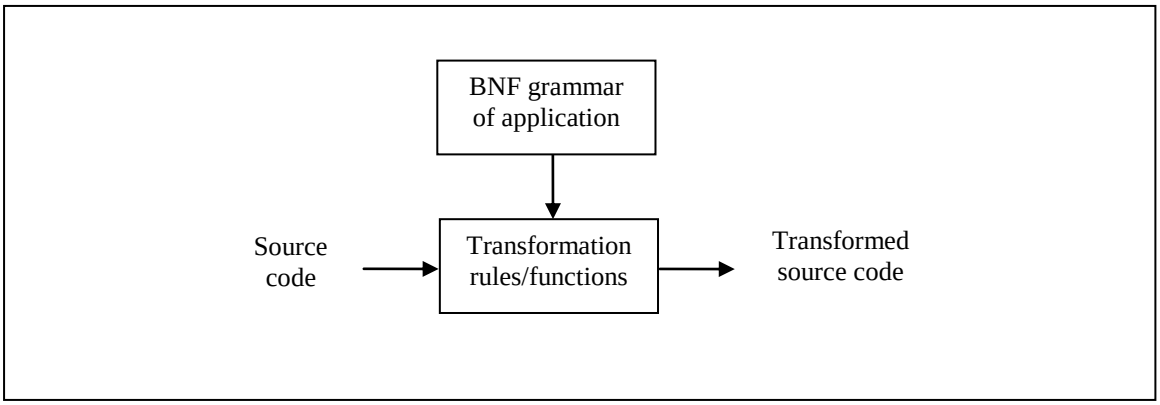

Figure 3.3: Source code transformation using TXL program

3.1.1 Vulnerable statements marking

In this step of DESERVE, all vulnerable statements are marked. By vulnerable statements, we refer to those statements whose execution can cause vulnerability exploitations. A vulnerable statement contains specific function calls or maintains certain patterns. The function calls and the patterns can vary from language to language. For BOF vulnerability in C application, a vulnerable statement contains C library function calls which modify program's memory space. For SQLI, a vulnerable statement can be a database access function call. A HTML generating function can be vulnerable for an XSS attack.

To identify vulnerable statements and mark them properly, a TXL program needs the knowledge of the syntactic structure of the implementation language (grammar). The grammar is needed by all three steps of the framework (See Figure 3.2). Grammars for popular languages are available in TXL's project website. We apply TXL functions and

rules for marking vulnerable statements. TXL transformation rules need the language grammar to mark the vulnerable statements.

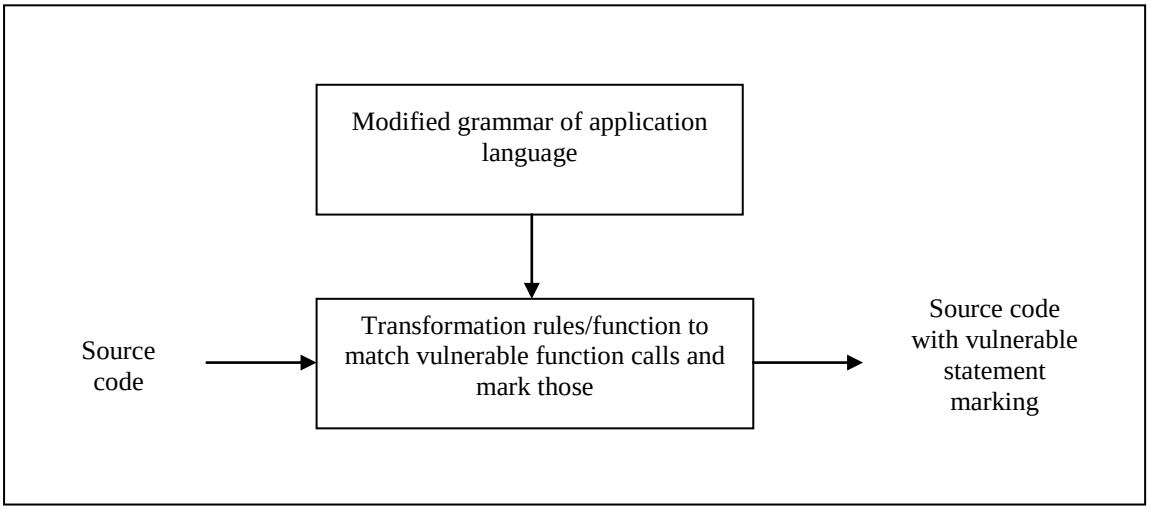

Figure 3.4: Vulnerable statement marking

Vulnerable statements are marked by adding XML like opening tag <*prblvulStmnt*> and closing tag <*/prblvulStmnt*>. Figure 3.4 shows the vulnerable statement marking procedure. Using the modified grammar, TXL transformation rules mark vulnerable statements with XML like tags. We enhance the grammar so that it accepts XML like tags.

3.1.2 Static backward slicing to identify exploitable vulnerable statements

DESERVE computes the slice of each marked vulnerable statement. TXL transformation rules are applied to determine the slice. The backward slice of a program point has only those statements which have an effect on it. The first parameter of slicing criteria is program point *p*. In our case, *p* is a marked vulnerable statement. The other

parameter *V* is a subset of program variables. Initially, *V* has the variables used in *p*. If the slice of a vulnerable statement has any input taking statement or *V* has any parameter of the executing function, we consider that vulnerable statement exploitable. By *exploitable vulnerable statemen*t, we mean those vulnerable statements whose executions are being influenced by any user input.

We apply TXL transformation rules to find out the slice. TXL rules add XML like opening tag <*runningvulStmnt*> and closing tag <*/runningvulStmnt*> to mark the program point currently under consideration. XML like opening tag <*mark*> and closing tag <*/mark*> are added to the statements which are in the slice.

Figure 3.5 shows the flowchart of this step. From the previous step, we already have all vulnerable statements marked with XML like opening tag <*prblvulStmnt*> and closing tag <*/prblvulStmnt*>. The program finds the slice of each vulnerable statement one by one, from top to bottom. In Step 1, TXL rules replace the first pair of <*prblvulStmnt*> and <*/prblvulStmnt*> tags by <*runningvulStmnt*> and <*/runningvulStmnt*>. After that, TXL program tracks back to the previous statement of the application's source code.

In Step 3, the program checks if the statement is an assignment statement or variable modification statement. In Step 4, it finds out if the modified expression has any variable used in the current slice. If yes, it marks statement with <*mark*><*/mark*>. When the TXL program hits the function top by backtracking, it advances to Step 7.

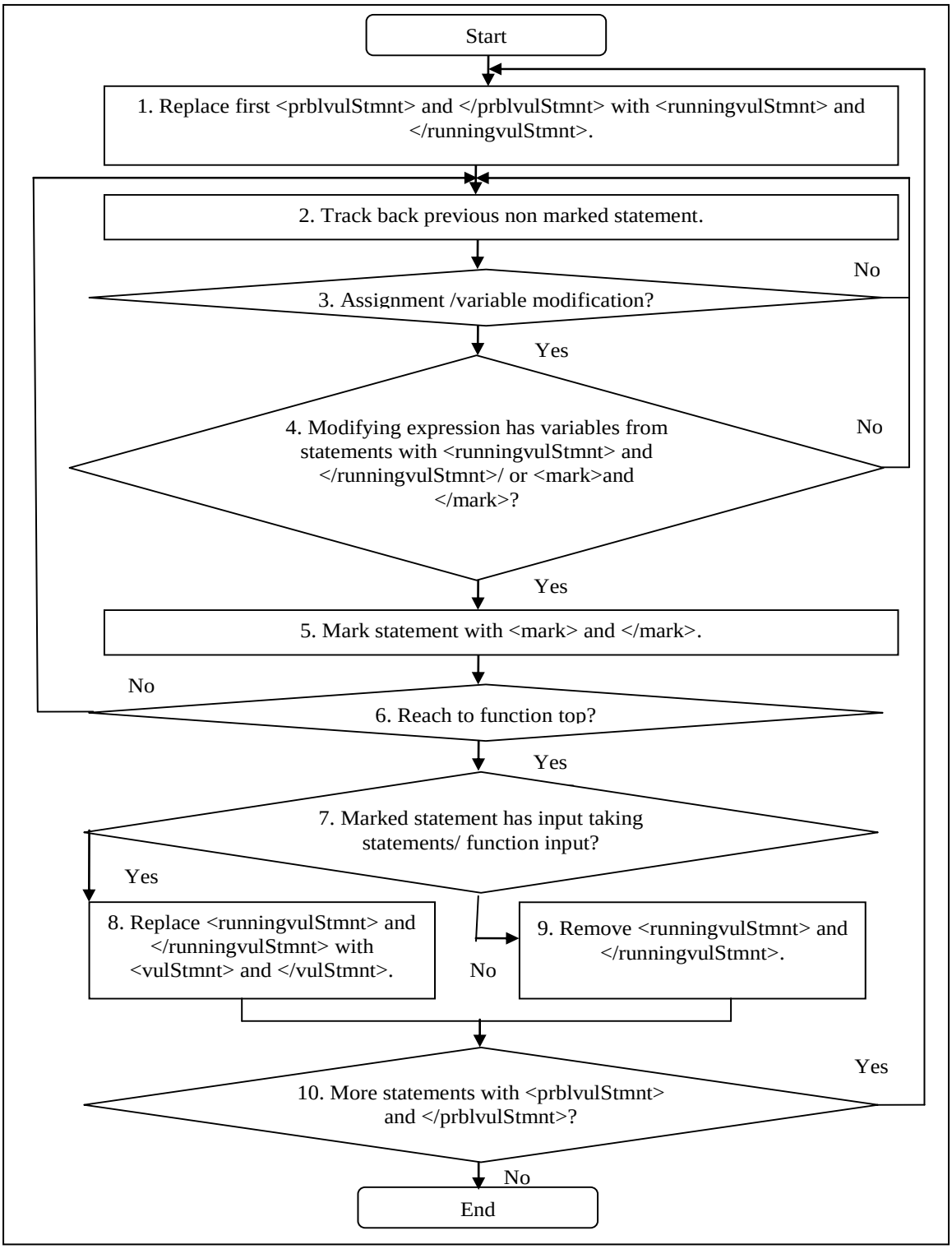

Figure 3.5: Program slicing flowchart

In Step 7, TXL program checks if the slice has any input taking statement or parameter of the executing function. If the answer is yes, it replaces <*runningvulStmnt*> and <*/runningvulStmnt* > by <*vulStmnt*> and <*/vulStmnt*>. We add XML like opening tag <*vulStmnt*> and closing tag <*/vulStmnt*> to indicate exploitable vulnerable statements.

3.1.3 Source code instrumentation

In this phase, DESERVE inserts code segment at exploitable vulnerable statements to make an application capable of self-monitoring. The inserted code segment mainly does the following three tasks to monitor itself:

1. The embedded monitor arranges a test environment to execute an exploitable vulnerable statement. This test environment may vary from language to language and vulnerability to vulnerability. For example, for BOF in C programming language a test environment can be a new forked process. For SQLI, a test environment can be a replica database. For XSS, a test environment can be a virtual representation of a HTML document.

2. The monitor tests the exploitable vulnerable statement for current input. We consider this testing as *runtime testing*. If the test implies no exploitation, it returns safe status with additional value to the original program. Otherwise, it returns to the original program with an exploitation notification. The testing of the exploitable statement depends on the vulnerability. To test SQLI and XSS, the monitor embedded by DESERVE matches the outcome of the exploitable

statement execution with the outcome of a sample safe execution of that statement. For BOF, our monitor tests the buffer size and content after the execution of the exploitable statement.

3. The original program waits for the test completion. The exploitable vulnerable statements are executed only if the tests return with safety notifications.

The instrumentation is also done based on the transformation rules in TXL. For each statement with XML like opening tag <*vulStmnt*> and closing tag <*/vulStmnt*>, the TXL program inserts additional code. The inserted code varies from language to language. However, for a specific vulnerability of a specific language, the inserted code structure is identical. This circumstance makes the automation of code instrumentation possible using a TXL program.

3.2 Monitoring

The output of the framework is instrumented source code. This instrumented source code is capable of self monitoring. The behavior of the instrumented application remains the same during runtime, except for the exploitable vulnerable statements. Figure 3.6 presents the flowchart of the monitoring behavior at any exploitable statement point. Whenever the execution reaches to any of the exploitable statements, a test environment is initiated (Step 1). In Step 2, the statement is executed in the test environment. The test environment depends on the type of vulnerabilities and the implementation language. We define the test environment for BOF, SQLI, and XSS. The detail implementations are provided in the next three chapters. In Step 3, the outcome of

the execution is used to determine an exploitation. The way of determining an exploitation also depends on the type of the vulnerability. The details are provided in the following chapters. If there is no exploitation, the original program executes the statement at Step 4.

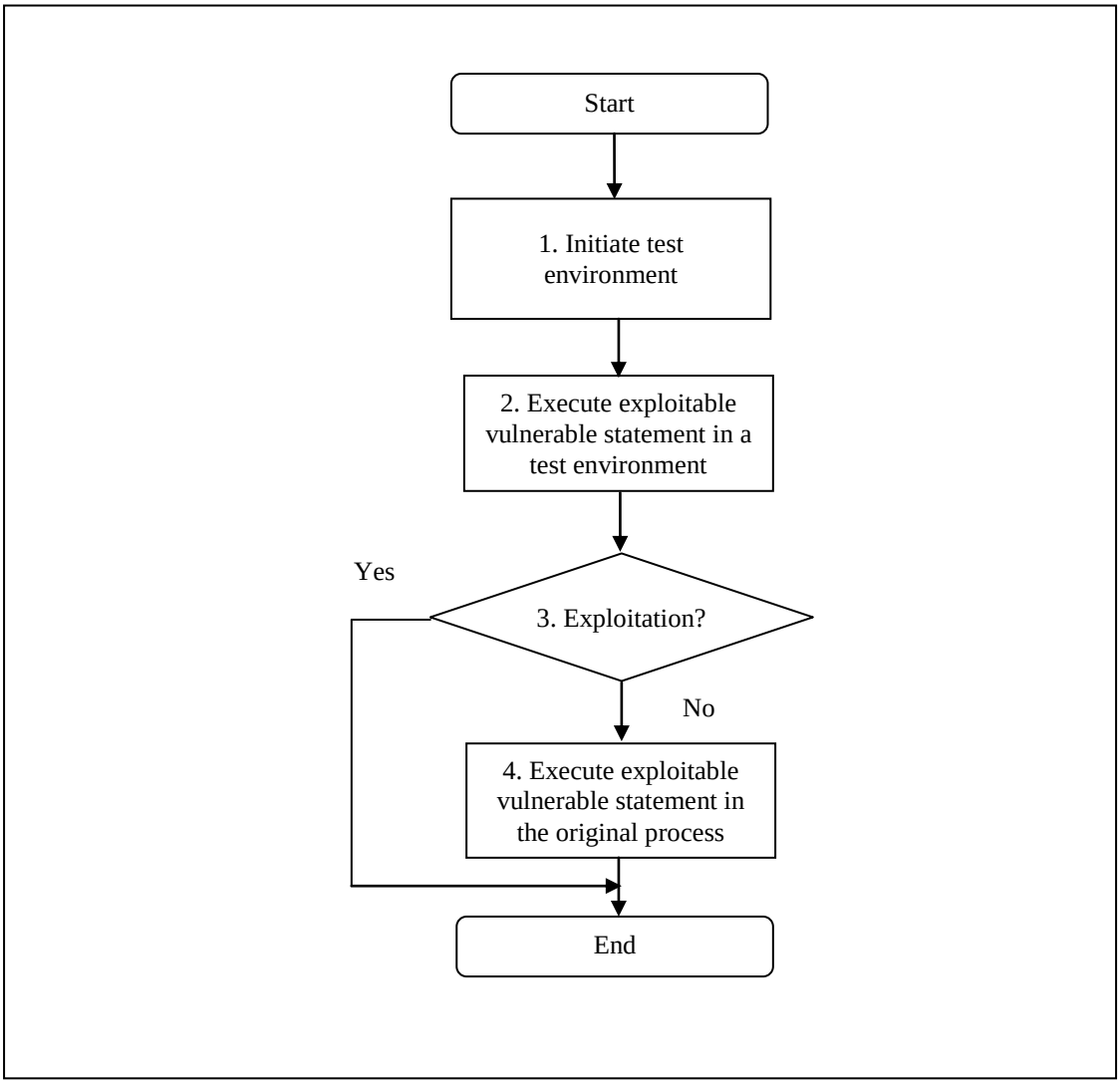

Figure 3.6: Monitoring

3.3 Summary

In this chapter, we have provided the detail description of our monitor embedding framework DESERVE. We have discussed properties, attributes, inputs, outputs, and implementation guidelines of the framework. We have defined the working principle and the implementations of the three steps of DESERVE. We have also discussed the monitoring behavior for attack detection. We will present the framework implementation and the monitoring technique for BOF, SQLI, and XSS vulnerabilities in the next three chapters.

Chapter 4

Buffer Overflow Monitoring

In the previous chapter, we have presented the generalized view of our proposed monitor embedding framework. We have discussed the steps of the framework and monitoring behavior. We have provided guidelines to implement the framework. To evaluate the effectiveness of DESERVE, we implement the framework for three known vulnerabilities. In this chapter, we explain the framework implementation for BOF vulnerability detection. We discuss the inputs, the outputs, and the working principle of each step of the framework for BOF monitoring. We describe the embedded monitor to detect BOF exploitations. To evaluate this BOF monitor, we instrument the source code of three open source programs wu-ftpd-2.6.2, gzip-1.3.12, and crafty-23.4 [65-67]. We also evaluate the effectiveness of DESERVE and discuss the performance of the embedded monitor.

4.1 Monitor embedding framework

In this section, we discuss the framework for BOF in C/C++ language based on the general framework described in Chapter 3. We discuss the three steps of the framework for BOF monitoring. We present the inputs, the outputs, the way of implementation, and examples for each step.

Table 4.1: Vulnerable statement patterns repository for BOF in C/C++

Function	Initial variable set V for slicing
char * strcpy (char * dst, const char * src)	src
char * strcat (char * dst, const char * src)	src
char * gets (char * str)	str
int sprintf (char * str, const char * format, ...)	str
int vsprintf (char * str, const char * format, va_list arg)	str
int scanf (const char * format, ...)	The variable list
int sscanf (const char * str, const char * format, ...)	The variable list
arr[a]=b	a

4.1.1 Vulnerable statements marking

In this step, the monitor embedding framework marks all vulnerable statements. By vulnerable statements, we indicate those statements which can cause a BOF. C/C++ has several vulnerable library functions. Table 4.1 shows a subset of vulnerable functions in C/C++ that we consider to evaluate our approach. By adding entries to the vulnerable statement pattern repository, we can easily extend our framework to consider other library functions. All of these functions modify buffer contents. In other words, these functions can cause a BOF. The vulnerable functions/statement patterns shown in Table 4.1 are stored in the vulnerable statement pattern repository (see Figure 3.2).

To identify vulnerable statements and mark those properly, we need the knowledge of the syntactic structure of the implementation language of the application under study. We employ TXL to implement the grammar and

marking.

```
1 int mapping_chdir(char *orig_path)
2 {
3    int ret;
4    char *sl, *path;
5
6    strcpy(old_mapped_path, mapped_path);
7    path = &pathspace[0];
8    strcpy(path, orig_path);
9
10   /* / at start of path, set the start of the mapped_path to / */
11   if (path[0] == '/') {
12        mapped_path[0] = '/';
13        mapped_path[1] = '\0';
14        path++;
15   }
16
17   while ((sl = strchr(path, '/'))) {
18        char *dir;
19        dir = path;
20        *sl = '\0';
21        path = sl + 1;
22        if (*dir)
23            do_elem(dir);
24        if (*path == '\0')
25            break;
26   }
27   if (*path)
28        do_elem(path);
29
30   if ((ret = chdir(mapped_path)) < 0) {
31        strcpy(mapped_path, old_mapped_path);
32   }
33
34   return ret;
35 }
```

Figure 4.1: *mapping_chdir* of wu-ftpd-2.6.2

Figure 4.1 shows function *mapping_chdir* in ftpd.c of an ftp server: wu-ftpd-2.6.2. After the first step, vulnerable statements are marked using XML like opening tag *<prblvulStmnt>* and closing tag *</prblvulStmnt>*. Figure 4.2 shows the output after marking. The vulnerable statements at Lines 6, 8, and 31 are shown in grey background

in this figure. TXL rules/functions insert these tags into appropriate places based on the modified C++ grammar written in TXL.

```
1 int mapping_chdir(char *orig_path)
2 {
3   int ret;
4   char *sl, *path;
5
6   <prblvulStmnt>strcpy(old_mapped_path, mapped_path); </prblvulStmnt>
7   path = &pathspace[0];
8   <prblvulStmnt>strcpy(path, orig_path); </prblvulStmnt>
9
10   /* / at start of path, set the start of the mapped_path to / */
11   if (path[0] == '/') {
12        mapped_path[0] = '/';
13        mapped_path[1] = '\0';
14        path++;
15   }
16
17   while ((sl = strchr(path, '/'))) {
18        char *dir;
19        dir = path;
20        *sl = '\0';
21        path = sl + 1;
22        if (*dir)
23            do_elem(dir);
24        if (*path == '\0')
25            break;
26   }
27   if (*path)
28        do_elem(path);
29
30   if ((ret = chdir(mapped_path)) < 0) {
31        <prblvulStmnt>strcpy(mapped_path, old_mapped_path); </prblvulStmnt>
32   }
33
34   return ret;
35 }
```

Figure 4.2: *mapping_chdir* of wu-ftpd-2.6.2 after vulnerable statements marking

4.1.2 Static backward slicing to identify exploitable vulnerable statements

DESERVE computes the slice of each marked vulnerable statement. TXL transformation rules are applied to determine the slice. The first parameter of slicing criteria is a program point *p*. In our case, *p* is a marked vulnerable statement. The other parameter *V*, a subset of program variables, depends on the type of vulnerable statement. The second column of Table 4.1 shows the initial value of *V* for each type of vulnerable statement. If the slice of a vulnerable statement has any input taking statement or *V* has any input parameter of the executing function, we consider that vulnerable statement exploitable. In our framework implementation for BOF, we consider *scanf* and *gets* as input taking functions. By exploitable vulnerable statements, we mean vulnerable statements whose executions are influenced by a user input.

Figures 4.3(a)-(c) show the slices for each vulnerable statement of function *mapping_chdir*. The statements in a slice are enclosed with XML like opening tag <*mark*> and closing tag </*mark*>. As the slices of the first (Figure 4.3(a)) and the third (Figure 4.3(c)) vulnerable statements neither have an input taking statement nor use any input parameter of the executing function, these two statements are not exploitable. The framework deletes the XML tag around these statements. However, the second vulnerable statement (Figure 4.3(b)) uses an input parameter of the executing function *orig_path*. Thus, our framework marks it as exploitable by changing the XML tag to <*vulStmnt*>.

```
1 int mapping_chdir(char *orig_path)
2 {
3    int ret;
4    char *sl, *path;
5
6    <prblvulStmnt>strcpy(old_mapped_path,
mapped_path); </prblvulStmnt>
7    path = &pathspace[0];
8    <prblvulStmnt>strcpy(path, orig_path);
</prblvulStmnt>
9
10    /* / at start of path, set the start of the
mapped_path to / */
11    if (path[0] == '/') {
12         mapped_path[0] = '/';
13         mapped_path[1] = '\0';
14         path++;
15    }
16
17    while ((sl = strchr(path, '/'))) {
18         char *dir;
19         dir = path;
20         *sl = '\0';
21         path = sl + 1;
22         if (*dir)
23              do_elem(dir);
24         if (*path == '\0')
25              break;
26    }
27    if (*path)
28         do_elem(path);
29
30    if ((ret = chdir(mapped_path)) < 0) {
31         <prblvulStmnt>strcpy(mapped_path,
old_mapped_path); </prblvulStmnt>
32    }
33
34   return ret;
35 }
```

a. Slice of Line 6

```
1 int mapping_chdir(char *orig_path)
2 {
3    int ret;
4    char *sl, *path;
5
6    <prblvulStmnt>strcpy(old_mapped_path,
mapped_path); </prblvulStmnt>
7    path = &pathspace[0];
8    <prblvulStmnt>strcpy(path, orig_path);
</prblvulStmnt>
9
10    /* / at start of path, set the start of the
mapped_path to / */
11    if (path[0] == '/') {
12         mapped_path[0] = '/';
13         mapped_path[1] = '\0';
14         path++;
15    }
16
17    while ((sl = strchr(path, '/'))) {
18         char *dir;
19         dir = path;
20         *sl = '\0';
21         path = sl + 1;
22         if (*dir)
23              do_elem(dir);
24         if (*path == '\0')
25              break;
26    }
27    if (*path)
28         do_elem(path);
29
30    if ((ret = chdir(mapped_path)) < 0) {
31         <prblvulStmnt>strcpy(mapped_path,
old_mapped_path); </prblvulStmnt>
32    }
33
34   return ret;
35 }
```

b. Slice of Line 8

```
1 int mapping_chdir(char *orig_path)
2 {
3    int ret;
4    char *sl, *path;
5
6 <prblvulStmnt>strcpy(old_mapped_path,
mapped_path); </prblvulStmnt>
7    path = &pathspace[0];
8    <prblvulStmnt>strcpy(path, orig_path);
</prblvulStmnt>
9
10    /* / at start of path, set the start of the
mapped_path to / */
11    if (path[0] == '/') {
12          <mark>mapped_path[0] = '/';</mark>
13          <mark>mapped_path[1] = '\0';</mark>
14          path++;
15    }
16
17    while ((sl = strchr(path, '/'))) {
18          char *dir;
19          dir = path;
20          *sl = '\0';
21          path = sl + 1;
22          if (*dir)
23              do_elem(dir);
24          if (*path == '\0')
25              break;
26    }
27    if (*path)
28          do_elem(path);
29
30    if ((ret = chdir(mapped_path)) < 0) {
31          <prblvulStmnt>strcpy(mapped_path,
old_mapped_path); </prblvulStmnt>
32    }
33
34   return ret;
35 }
```

c. Slice of Line 31

```
1 int mapping_chdir(char *orig_path)
2 {
3    int ret;
4    char *sl, *path;
5
6    strcpy(old_mapped_path, mapped_path);
7    path = &pathspace[0];
8    <vulStmnt>strcpy(path,
orig_path);</vulStmnt>
9
10    /* / at start of path, set the start of the
mapped_path to / */
11    if (path[0] == '/') {
12          mapped_path[0] = '/';
13          mapped_path[1] = '\0';
14          path++;
15    }
16
17    while ((sl = strchr(path, '/'))) {
18          char *dir;
19          dir = path;
20          *sl = '\0';
21          path = sl + 1;
22          if (*dir)
23              do_elem(dir);
24          if (*path == '\0')
25              break;
26    }
27    if (*path)
28          do_elem(path);
29
30    if ((ret = chdir(mapped_path)) < 0) {
31          strcpy(mapped_path,
old_mapped_path);
32    }
33
34   return ret;
35 }
```

d. Exploitable statements marking

Figure 4.3: *mapping_chdir* of wu-ftpd-2.6.2 after slicing

4.1.3 Source code instrumentation

In this phase, DESERVE inserts a code segment at an exploitable vulnerable statement to make the application capable of self-monitoring. The embedded monitor needs to arrange a test environment to execute an exploitable vulnerable statement. For BOF exploitation detection in C/C++, we consider a forked process as a test environment. The new forked process has the exact copy of the address space of the original process. We use operating system pipe for necessary inter process communications.

The test process executes the exploitable vulnerable statement. As the address space of the original process and the test process are identical, after effect of the execution would be exactly the same. After the execution, the test process checks the following two conditions to detect an attack:

1. If the current buffer length is greater than the allocated size for the buffer.
2. If the statement execution alters the adjacent memory cell of the buffer.

If any of the above conditions is true, the test process sends a failure notification to the original process. Otherwise, it sends the success notification and pushes necessary data into the pipe.

The original process waits for the test completion. If it receives a success notification, it collects necessary values from the pipe. Then, the original process executes the vulnerable statement safely.

DESERVE already has marked all exploitable statements of the source code with the XML like opening tag <*vulStmnt*> and the closing tag <*/vulStmnt*>. To embed a BOF monitor, this step of DESERVE inserts the monitoring code at a tagged point. A TXL program is developed to instrument the source code automatically. We have discussed the TXL program in Chapter 3. The automation of the instrumentation becomes possible as the inserted code is always the same for a specific vulnerable function.

```
1 int mapping_chdir(char *orig_path)
2 {
3    int ret;
4    char *sl, *path;
5
6    strcpy(old_mapped_path, mapped_path);
7    path = &pathspace[0];
8    /*instrumentation starts*/
9 fflush (stdout);
10    pidpid = fork ();
11    if (pidpid == 0) {
12        temp = path [sizeof (path)];
13        strcpy (path, orig_path);
14        if (temp != path [sizeof (path)]) {
15            return 1;
16        }
17        if (strlen (path) > sizeof (path)) {
18            return 1;
19        }
20        return 0;
21    }
22    wait (&status);
23    if (status == 0) strcpy (path, orig_path);
24
25   /* / at start of path, set the start of the
mapped_path to / */
26    if (path[0] == '/') {
27          mapped_path[0] = '/';
28          mapped_path[1] = '\0';
29          path++;
30  }
31
32    while ((sl = strchr(path, '/'))) {
33          char *dir;
34          dir = path;
35          *sl = '\0';
36          path = sl + 1;
37          if (*dir)
38              do_elem(dir);
39          if (*path == '\0')
40              break;
41    }
42    if (*path)
43          do_elem(path);
44
45    if ((ret = chdir(mapped_path)) < 0) {
46          strcpy(mapped_path,
old_mapped_path);
47    }
48
49    return ret;
50 }
```

Figure 4.4: *mapping_chdir* of wu-ftpd-2.6.2 after instrumentation

```
1 int SockPrintf(FILE *sockfp, char *format,...)
2 {
3    va_list ap;
4    char buf[32768];
5
6 /*instrumentation starts*/
7 pipe(fdfd);
8 fflush (stdout);
9    pidpid = fork ();
10    if (pidpid == 0) {
11        temp = buf [sizeof (buf)];
12        va_start(ap, format);
13        vsprintf (buf, format, ap);
14        va_end(ap);
15        if (temp != wildcard [sizeof (buf)]) {
16            return 1;
17        }
18        if (strlen (buf) > sizeof (buf)) {
19            return 1;
20        }
21       close(fd [0]);
22       write (fd [1], buf, (strlen (buf) + 1));
23 return 0;
24    }
25    wait (& status);
26    close(fdfd [1]);
27    if (status == 0) {

28       close (fd [1]);
29       read (fd [0], buf, sizeof (buf));
30    }
31
32    return SockWrite(buf, 1, strlen(buf), sockfp);
33 }
```

Figure 4.5: *sockPrintf* of wu-ftpd-2.6.2 after instrumentation

Figure 4.4 shows the outcome after the code instrumentation of function *mapping_chdir*. This is the final output of the monitoring framework. This instrumented source code is capable of self-monitoring. Lines 9-23 are the inserted source code. Lines 11-21 are executed in a test process. At Line 10, the exploitable statement is executed.

BOF detection is done in Lines 14-19. At Line 22, the original program waits for the test process. If it receives a success status, it executes the exploitable statement at Line 23.

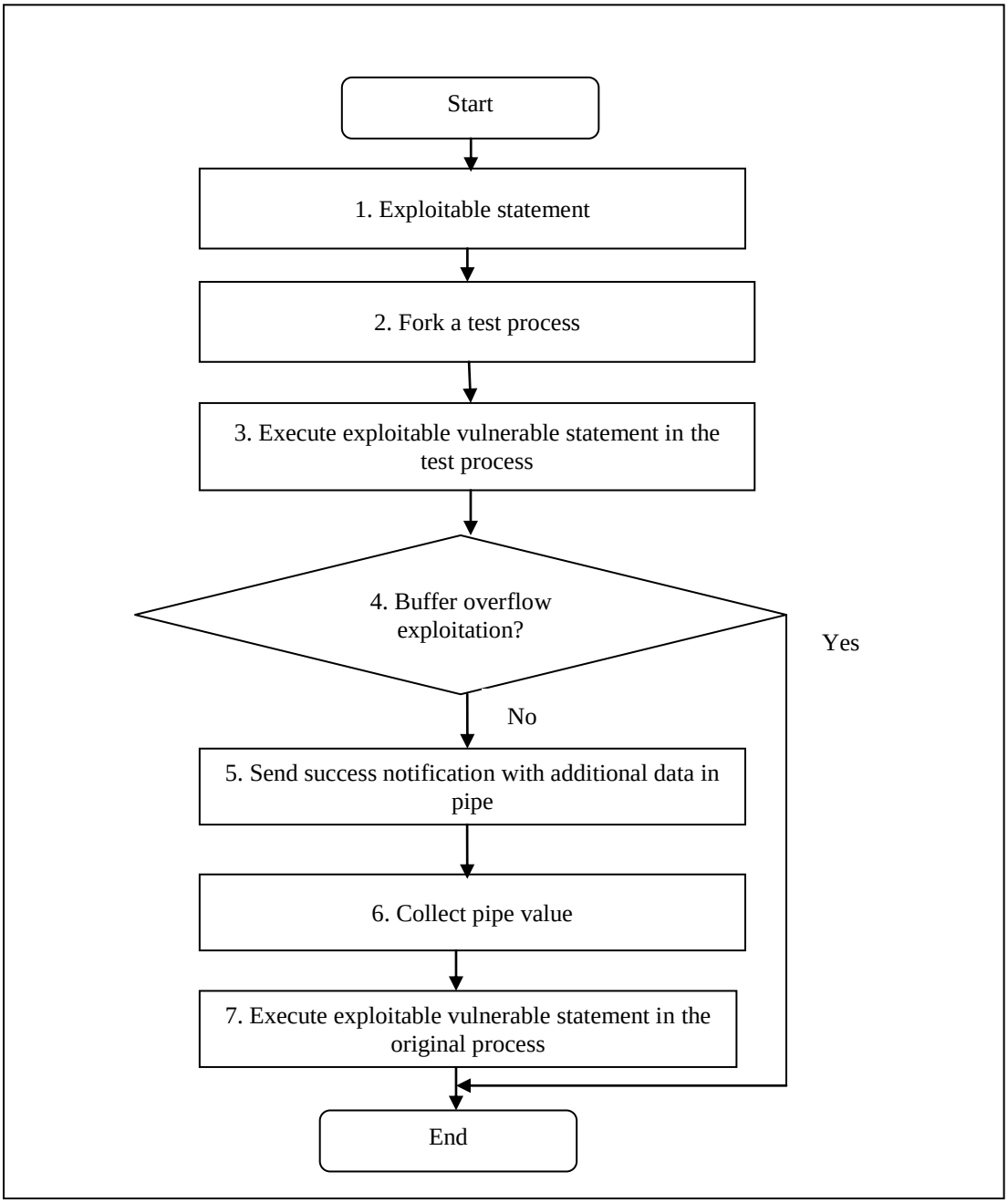

Figure 4.6: Buffer overflow monitoring

Figure 4.5 shows the instrumented *sockPrintf* function. In this instrumentation, the test environment returns the safety notification with additional value using a pipe. The pipe is declared at Line 7. The test process writes into the pipe at Line 22. The original process reads the pipe value at Line 29.

4.2 Monitoring

Figure 4.6 shows the BOF monitoring flowchart of the monitor embedded by DESERVE. Whenever any program execution reaches up to an exploitable vulnerable statement (at Step 2), a test process is forked. The forked process has the identical address space and properties of the original program. At Step 3 of this flowchart, the statement is executed in the test process. The test process detects the exploitable statement using the conditions described in Subsection 4.1.3. If there is no exploitation, the test process sends back a success notification with additional data in pipe. We find at Step 6 that the original process collects the pipe value. At Step 7, the original process executes the statement. If Step 4 indicates an exploitation, Steps 5-7 are skipped by the monitor.

4.3 Evaluation

We choose applications which are widely used and known for having BOF vulnerabilities. We evaluate our framework using three applications: wu-ftpd-2.6.2, Gzip-1.3.12 and Crafty-23.4. These applications are implemented in C. Wu-ftpd is a free ftp server for UNIX like system. It has several known vulnerabilities including BOF.

Gzip stands for GNU zip. It is used for file compression and decompression. Crafty is an open source chess program.

After vulnerable statement marking, we find that wu-ftpd-2.6.2 has 431 vulnerable statements; Gzip-1.3.12 has 513 vulnerable statements; and Crafty-23.4 has 2769 vulnerable statements. Static backward slicing at the second step points out that in wu-ftpd only 23 statements out of 431 vulnerable statements are exploitable. For Gzip and Crafty these numbers are 14 and 43, respectively. The number of the exploitable vulnerable statements is 1.6-5.3 percent of the number of vulnerable statements. The individual break down for each vulnerable statement pattern is given in Table 4.2. Some exploitable vulnerable statements recognized by DESERVE are the known weak points of the applications, which are responsible for vulnerabilities like CVE-2003-1327, CVE-1999-0878, CVE-2003-0466, CVE-2001-1228, CVE-2003-1327 etc [63].

Table 4.2: Exploitable statements for BOF attack in wu-ftpd, Gzip, and Crafty

Applications	**Wu-ftpd**			**Gzip**			**Craft**		
Statement types	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement
Strcpy	125	17	13.6	21	4	19	219	8	3.7
Strcat	31	3	9.8	6	0	0	37	1	2.7
Gets	66	0	0	73	0	0	50	0	0
Sprintf	80	0	0	13	0	0	153	0	0
Vsprintf	2	2	100	0	0	0	0	0	0
Scanf	21	0	0	0	0	0	18	0	0
Sscanf	20	0	0	0	0	0	6	4	66.7
Array assignment	86	1	1.2	400	10	2.5	2286	30	1.3
Total	**431**	**23**	**5.3**	**513**	**14**	**2.72**	**2769**	**43**	**1.6**

We instrument the source code at exploitable vulnerable statements. The instrumentation increases the line of code.

Table 4.3 shows the data about additional line of code after the instrumentation. The overall line of code increase due to the instrumentation is 1.2-2.15 percent.

Table 4.3: Added line of code after code instrumentation for BOF attack

Applications	**Wu-ftpd**	**Gzip**	**Craft**
Line of Code	24,531	14,174	44,632
Line of code added	304	168	960
% increased	1.2	1.2	2.15

As we apply runtime testing to detect exploitation, the monitor has runtime overhead. The overhead is constant and introduces almost the same delay before each exploitable vulnerable statement execution. We use Dell optiplex 980, Intel core i7 @ 2.80 GHz quad core machine as the evaluation environment. With this system specification, for a safe input to an exploitable statement, the monitor introduces only 0.78 micro second delays. For a moderate size input, which introduces BOF, the monitor needs 2.5 milliseconds to handle the overflow.

4.4 Summary

In this chapter, we have presented the implementation of the framework for BOF monitoring. We have described the implementation, input, and output of each step of the framework with examples. We have discussed the vulnerable statements of C language for the BOF attack and designed the test environment for their runtime testing. The

monitoring of the BOF exploitation has been explained too. We have applied this framework on three well known open source programs and measured the number of vulnerable statements, exploitable statements, performance, and overheads for each program. The embedded monitor can successfully detect the known BOF vulnerabilities of these programs. In summary, we can say that the embedded monitor can successfully detect a BOF attack with minimal runtime overhead.

Chapter 5

SQL Injection Monitoring

In this chapter, we discuss about the DESERVE implementation for SQLI monitoring. The SQLI monitor embedded by DESERVE can monitor both first and second order SQLI attacks. We define the input and the output of the framework and the implementation detail of each step to embed an SQLI monitor. We describe the runtime testing environment for the SQLI monitoring and explain the behavior of the monitor. The framework is applied on three open source programs SchoolMate_v1.5.4, WebChess v1.0.0rc2, and FAQforge-1.3.2 [68-70]. Finally, the necessary statistics with respect to the effectiveness of DESERVE and the embedded monitor is provided. We present the detail about the framework implementation in Section 5.1 and discuss this SQLI monitoring in section 5.2. The evaluation of the monitor is discussed in Section 5.3.

5.1 Monitor embedding framework

We implement the framework for PHP web applications which use MySQL database server. We choose PHP applications to evaluate the framework because 75 percent of web applications are developed in PHP [58]. In 2010, 27.2 percent of total web vulnerabilities were in PHP web applications [59]. We choose MySQL database server as it is the most popular open source database server [60].

Other than an application source code, the framework needs two inputs: the PHP grammar and a list of PHP functions which are vulnerable for SQLI attacks. As we have mentioned in Chapter 3, we developed TXL programs to implement the three steps of DESERVE. The grammar is used by the TXL programs to analyze the syntactical structure of the source code. We list the vulnerable functions in PHP in Table 5.2. We describe the three steps of the framework implementation for SQLI monitoring in Sections 5.1.1-5.1.3.

5.1.1 Vulnerable statements marking

In this step, the monitor embedding framework marks all vulnerable statements. By vulnerable statements, we indicate those statements which can cause an SQLI. Database access functions are called to execute a SQL script by server side scripts. More precisely, database access functions are responsible for the execution of a malicious SQL script. Therefore, the statements which have a database access function call are considered vulnerable. Table 5.1 shows database access functions in PHP for different database servers. However, we consider the applications which use MySQL database. Table 5.2 shows MySQL database access functions in PHP. A statement which has any database access functions listed in Table 5.2 is considered vulnerable.

We apply TXL programming language to implement the grammar and marking. Figure 5.1 shows two code snippets from a web based chess program: WebChess v1.0.0rc2. After the first step, vulnerable statements are marked using XML like opening tag <*prblvulStmnt*> and closing tag <*/prblvulStmnt*>. Figure 5.2 shows the modified

snippets after marking. The vulnerable statement is at Line 2 in Figure 5.2(a). Lines 1 and 3 of Figure 5.2(b) are vulnerable. These statements are shown in grey background in the figure. TXL rules/functions insert these tags into proper places using modified PHP grammar. We modify the grammar so that it can accept the XML like tag.

Table 5.1: PHP database access functions for different database servers

Function	Database Server Type
resource mysql_db_query (string $database , string $query [, resource $link_identifier])	MySQL
resource mysql_unbuffered_query (string $query [, resource $link_identifier])	MySQL
resource mysql_query (string $query [, resource $link_identifier])	MySQL
mixed mssql_execute (resource $stmt [, bool $skip_results = false])	MSSQL
mixed mssql_query (string $query [, resource $link_identifier [, int $batch_size = 0]])	MSSQL
bool oci_execute (resource $statement [, int $mode = OCI_COMMIT_ON_SUCCESS])	OCI8
resource pg_execute ([resource $connection], string $stmtname , array $params)	PostgreSQL
resource pg_query ([resource $connection], string $query)	PostgreSQL
bool pg_send_execute (resource $connection , string $stmtname , array $params)	PostgreSQL

5.1.2 Static backward slicing to identify exploitable vulnerable statements

By exploitable vulnerable statements for an SQLI attack, we mean those vulnerable statements whose executions are being influenced by a user input (immediate or stored). In this step, DESERVE employs static backward slicing to identify the exploitable statements. Our framework computes the slice of each vulnerable statement. As we mentioned in Chapter 2, to determine a slice, DESERVE needs to know the program point *p* and the subset of program variables *V*. In our case, *p* is a vulnerable

statement which has been marked in the previous step. The other parameter *V* which is the subset of program variables depends on the type of a vulnerable statement. Table 5.2 shows the initial value of *V* for each type of vulnerable statements.

Table 5.2: Vulnerable statement patterns repository for SQLI attack

Function	**Initial variable set V for slicing**
mysql_db_query (string $database , string $query [, resource $link_identifier])	$query
resource mysql_unbuffered_query (string $query [, resource $link_identifier])	$query
resource mysql_query (string $query [, resource $link_identifier])	$query

As a code injection attack, SQLI can be divided into first and second order attack. A first order attack occurs when an input from a user is used in a vulnerable statement. On the other hand, a stored input causes a second order attack. For a first order attack an input comes from a user, and for a second order attack an input comes from a database.

If the slice of a vulnerable statement uses any user input or *V* has any input parameter of the executing function, we consider that vulnerable statement exploitable by a first order attack. Normally a web application receives inputs from a user via HTTP GET/POST methods. PHP program can access these inputs using array *$_GET/$_POST*. Another way of receiving an input from a user is browser cookie. Cookie values can be found in array *$_COOKIE*. Array *$_REQUEST* by default contains the contents of *$_GET*, *$_POST*, and *$_COOKIE*. We consider that a statement uses user inputs, if it uses any of these arrays.

```
1 $tmpQuery = "SELECT playerID FROM " . $CFG_TABLE[players] . " WHERE nick =
'".$_POST['txtNick']."'";
2 $existingUsers = mysql_query($tmpQuery);
```

a. Snippet from mainmenu.php

```
1 $tmpMaxTime = mysql_query("SELECT Max(timeOfMove) FROM " . $CFG_TABLE[history] . "
WHERE gameID = ".$_SESSION['gameID']);
2 $maxTime = mysql_result($tmpMaxTime,0);
3 $moves = mysql_query("SELECT * FROM " . $CFG_TABLE[history] . " WHERE gameID =
".$_SESSION['gameID']." AND timeOfMove = '$maxTime'");
```

b. Snippet from undo.php

Figure 5.1: Code snippet from WebChess v1.0.0rc2

```
1 $tmpQuery = "SELECT playerIDFROM " . $CFG_TABLE[players] . " WHERE nick =
'".$_POST['txtNick']."'";
2 <prblvulStmnt> $existingUsers = mysql_query($tmpQuery);</ prblvulStmnt>
```

a. Snippet adopted from mainmenu.php

```
1 <prblvulStmnt> $tmpMaxTime = mysql_query("SELECT Max(timeOfMove) FROM " .
$CFG_TABLE[history] . " WHERE gameID = ".$_SESSION['gameID']);</prblvulStmnt>
2 $maxTime = mysql_result($tmpMaxTime,0);
3 <prblvulStmnt>$moves = mysql_query("SELECT * FROM " . $CFG_TABLE[history] . " WHERE
gameID = ".$_SESSION['gameID']." AND timeOfMove = '$maxTime'");</prblvulStmnt>
```

b. Snippet adopted from undo.php

Figure 5.2: Code snippet from WebChess v1.0.0rc2 after vulnerable statements marking

For second order attacks, a slice has statements which collect information from the database. If the slice of a vulnerable statement has a database access function, we

consider the vulnerable statement exploitable. The list of PHP database access functions for accessing MySQL database server is given in the first column of Table 5.2.

Figure 5.3 shows the slice of Figure 5.1(a) for the first order attack. The statements in a slice are enclosed within XML like opening tag <*mark*> and closing tag </*mark*>. A vulnerable statement at Line 2 uses a variable *$tmpQuery*. The statement at Line 1, which is also in the slice, updates variable *$tmpQuery* using array *$_POST*. Therefore, the vulnerable statement at Line 2 can be exploitable. Therefore, DESERVE marks it as exploitable by changing the XML tag to <*vulStmnt*>.

```
1 <mark>$tmpQuery = "SELECT playerID FROM " . $CFG_TABLE[players] . " WHERE nick =
'".$_POST['txtNick']."'";</mark>
2 <prblvulStmnt> $existingUsers = mysql_query($tmpQuery);</ prblvulStmnt >
```

Figure 5.3: Code snippet from WebChess v1.0.0rc2 after slicing

```
1 <prblvulStmnt> $tmpMaxTime =
mysql_query("SELECT Max(timeOfMove)
FROM " . $CFG_TABLE[history] . " WHERE
gameID =
".$_SESSION['gameID']);</prblvulStmnt>
2 $maxTime = mysql_result($tmpMaxTime,0);
3 <prblvulStmnt>$moves =
mysql_query("SELECT * FROM " .
$CFG_TABLE[history] . " WHERE gameID =
".$_SESSION['gameID']." AND timeOfMove =
'$maxTime'");</prblvulStmnt>
```

a. Slice of Line 1

```
1 <prblvulStmnt> $tmpMaxTime =
mysql_query("SELECT Max(timeOfMove)
FROM " . $CFG_TABLE[history] . " WHERE
gameID =
".$_SESSION['gameID']);</prblvulStmnt>
2 <mark>$maxTime =
mysql_result($tmpMaxTime,0);</mark>
3 <prblvulStmnt>$moves =
mysql_query("SELECT * FROM " .
$CFG_TABLE[history] . " WHERE gameID =
".$_SESSION['gameID']." AND timeOfMove =
'$maxTime'");</prblvulStmnt>
```

b. Slice of Line 3

Figure 5.4: Code snippet from WebChess v1.0.0rc2 after slicing

Figure 5.4 shows the slices for each vulnerable statement of the code snippet from Figure 5.1(b) for the second order attacks. The statements in a slice are enclosed within XML like opening tag <*mark*> and closing tag <*/mark*>. As the slice of the first (at Line 1) vulnerable statement does not have any input from a database, this statement is not exploitable. DESERVE deletes the XML tag around these statements. However, the second vulnerable statement (at Line 3) uses a variable *$maxTime*. This variable is fetched from the database (Lines 1-2). As a result, our framework marks it as exploitable by changing the XML tag to <*vulStmnt*>. Figure 5.5 shows the code snippet of Figure 5.1 after exploitable statement marking.

```
1 $tmpQuery = "SELECT playerID FROM " . $CFG_TABLE[players] . " WHERE nick =
'".$_POST['txtNick']."'";
2 <vulStmnt> $existingUsers = mysql_query($tmpQuery);</vulStmnt >
```

a. Snippet from mainmenu.php

```
1 $tmpMaxTime = mysql_query("SELECT Max(timeOfMove) FROM " . $CFG_TABLE[history] . "
WHERE gameID = ".$_SESSION['gameID']);
2 $maxTime = mysql_result($tmpMaxTime,0);
3 <vulStmnt>$moves = mysql_query("SELECT * FROM " . $CFG_TABLE[history] . " WHERE
gameID = ".$_SESSION['gameID']." AND timeOfMove = '$maxTime'");</vulStmnt>
```

b. Snippet from undo.php

Figure 5.5: Code snippet from WebChess v1.0.0rc2 after exploitable statements marking

5.1.3 Source code instrumentation

In this phase, DESERVE inserts code segment at an exploitable vulnerable statement to make the application capable of self-monitoring. It needs to arrange a test environment to execute exploitable vulnerable statements. To detect an SQLI attack, we consider a replica database in a test database server as a test environment. The new replica database and the original database have the identical data. In the worst case, malicious SQL code can delete database tables; delete the database; and shutdown the server. In our setup, any destructive SQL command is executed first on the test database. The embedded monitor can detect the attack and the SQL command is not executed in the original database. Thus, even for a malicious SQL script, the original application database remains untouched.

The monitor executes the exploitable database access functions on a replica database to detect an attack. If the statement executes safely, we execute the SQL in the original database. To identify a safe execution, we check the following conditions:

1. Does the execution generate any error?
2. Are the numbers of fetched, added, edited, and deleted data rows greater than the thresholds?
3. Are the numbers of fetched, added, edited, and deleted objects greater than the thresholds?

If any of the above conditions is false, the SQL execution is not safe and the SQL is not executed in the original database. For each exploitable statement, DESERVE needs

to know the threshold values. The *thresholds* are the maximum numbers of fetched, added, edited, and deleted data rows and objects which are returned from the database after the safe execution of an exploitable statement. DESERVE takes these values as inputs during the processing of a source code.

We create a stored procedure in the test database which takes SQL script as input and executes it. We name it *testExecution*. The stored procedure *testExecution* collects the fetched/added/edited/deleted data rows and objects information for the SQL script execution. This procedure also collects information regarding the errors in execution. Finally, *testExecution* returns all these information to the caller application. To execute the SQL script in the test database, a PHP application calls the stored procedure with the script and compares the return values with thresholds to detect an attack.

```
1 $tmpQuery = "SELECT playerID FROM " . $CFG_TABLE[players] . " WHERE nick =
'".$_POST['txtNick']."'";
2 $result=mysql_query('call testExecution($tmpQuery)', testDBCon);
3 $status=mysql_fetch_assoc($result);
4 If (status['error']==0 && status['fetch']==1&& status['affected']==0&&
status['status']==$DesiredStatus){
5 $existingUsers = mysql_query($tmpQuery);
6}
```

Figure 5.6: Code snippet of WebChess v1.0.0rc2 after instrumentation

The instrumentation is also done using the transformation rules in TXL. For each statement with XML like opening tag <*vulStmnt*> and closing tag <*/vulStmnt*>, TXL code inserts additional code. The inserted code structure is identical except the variable names. As a result, a TXL program can instrument the source code automatically.

Figure 5.6 is the modified code snippet of Figure 5.1(a) after code instrumentation. At Line 2, the code snippet calls the stored procedure *testExecution* to execute the SQL in the test database. The return values are put into variable *$status* in Line 3. The necessary comparisons are done at Line 4. Line 5 executes the SQL statement in the original database as the statement execution is safe.

```
1  CREATE PROCEDURE testExecution (sql varchar(2000))
2  BEGIN
3     PREPARE stmt FROM sql;
4     EXECUTE stmt;
5     Select ROW_COUNT() as affected, FOUND_ROWS() as fetch, mysql_errno() as error,
mysql_stat()as status;
6  END
```

Figure 5.7: Stored procedure testExecution

5.2 Monitoring

Figure 5.8 shows the SQLI monitoring flowchart of our monitor. Whenever any program execution reaches up to an exploitable vulnerable statement, the SQL script is executed in a test database. The test database is a replica of the original database. At Step 2 of this flowchart, the statement is executed in the test database. The exploitation is detected using the conditions described in Subsection 5.1.3. If there is no exploitation, the SQL statement is executed in the original database.

5.3 Evaluation

We evaluate DESERVE by embedding monitors into applications SchoolMate_v1.5.4, WebChess v1.0.0rc2, and FAQforge-1.3.2 [68-70]. All of these applications are developed in PHP. MySQL is used as the database server.

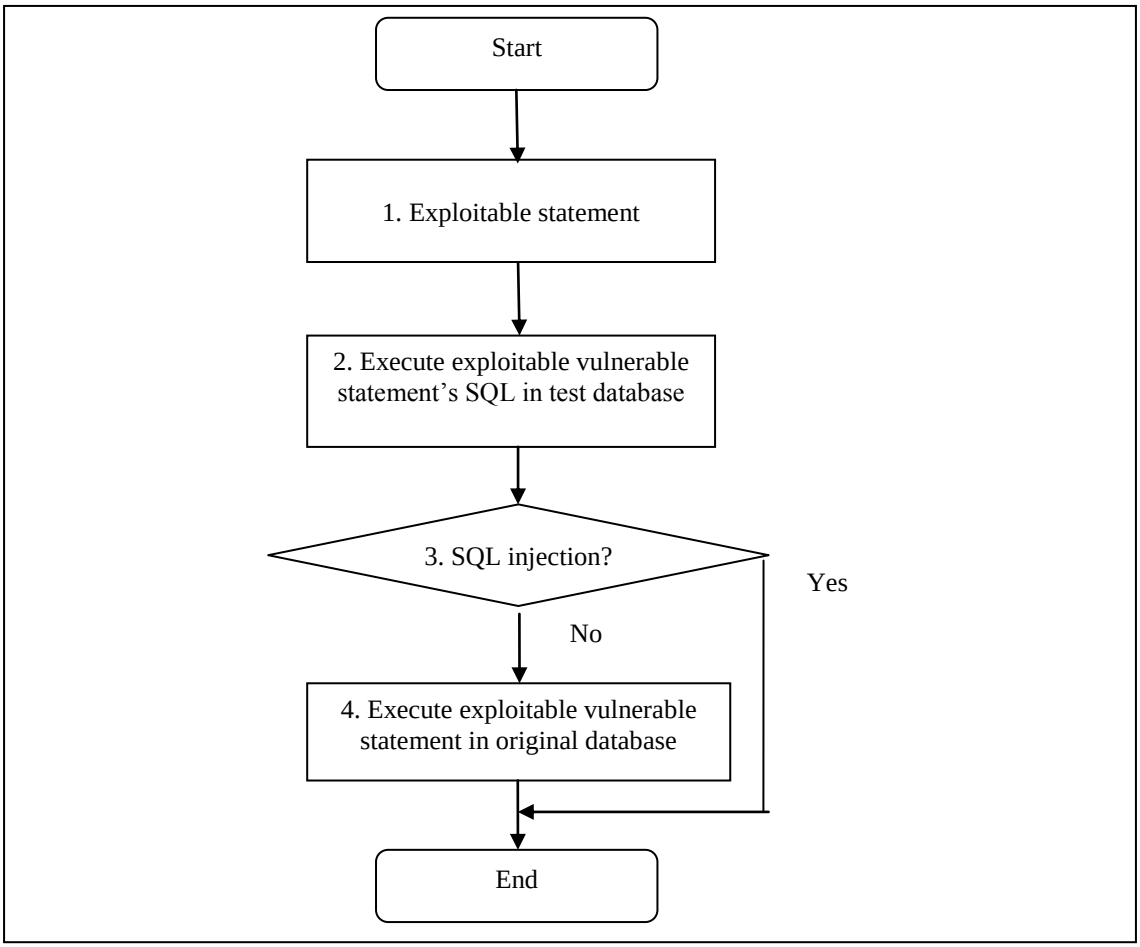

Figure 5.8: SQLI monitoring

SchoolMate is a web application for managing and sharing information about a school's activities [68]. A school administrator can add, edit, and delete the classes and the users. Teachers manage the assignments and grades. Students can view their information and parents can check their kids' progress. FAQforge is a documentation management tool. It is installed to create hierarchical documents manual [70]. WebChess is a web based application for playing chess [69]. It allows users to play chess with other users across the internet.

After the execution of the first step of DESERVE, all vulnerable statements of an application have been marked. For Schoolmate, the number of marked statement is 294. For WebChess and FAQforge these numbers are 121 and 33, respectively.

Static backward slicing at the second step of the framework recognizes and marks exploitable vulnerable statements. DESERVE recognizes exploitable statements for the first order and second order attacks separately. We identify that 73 statements in SchoolMate are marked as exploitable for the first order attacks. For WebChess and FAQforge these numbers are 53 and 15, respectively. The number of exploitable statements is 12.3-53 percent of the number of vulnerable statements. These findings are listed in Table 5.3.

Table 5.3: Exploitable statements in SchoolMate, WebChess, and FAQforge for first order SQLI attack

Applications	**SchoolMate**			**WebChess**			**FAQforge**		
Vulnerable functions	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement
mysql_query	294	73	25	121	53	44	33	15	45

The statistics about the exploitable statements for the second order SQLI attacks are presented in Table 5.4. SchoolMate, WebChess, and FAQforge have 186, 25, and 12 exploitable statements, respectively. The exploitable vulnerable statements are 21-63 percent of the vulnerable statements. The individual break down is presented in Table 5.4.

Table 5.4: Exploitable statements in SchoolMate, WebChess, and FAQforge for second order SQLI attack

Applications	**SchoolMate**			**WebChess**			**FAQforge**		
Vulnerable functions	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement
mysql_query	294	186	63	121	25	21	33	12	37

As DESERVE instruments source code for each exploitable statement, it increases the line of code of the application. Table 5.5 and Table 5.6 show the data about the additional line of code for the first and second order SQLI attack detections. For the first order attack monitoring, the size of the source code is increased by 3.3-4 percent. For the second order attack detection, 2.1-8.5 percent additional code is needed.

Table 5.5: Added line of code after code instrumentation for first order SQLI attack

Applications	**SchoolMate**	**WebChess**	**FAQforge**
Line of Code	6,554	3,640	1,045
Line of code added	219	159	45
% increased	3.3	4.4	4.3

Table 5.6: Added line of code after code instrumentation for second order SQLI attack

Applications	**SchoolMate**	**WebChess**	**FAQforge**
Line of Code	6,554	3,640	1,045
Line of code added	558	75	36
% increased	8.5	2.1	3.4

As the embedded monitor applies runtime testing to detect exploitations, the monitor has runtime overhead. The overhead is constant and it introduces same delay only before the execution of each exploitable vulnerable statement. For monitoring each

exploitable statement, an application needs an additional database access and some comparisons. If the time for accessing database once is *t* and the time for comparisons is *c*, theoretically, the execution time of each exploitable statement will be *2t+c*. Therefore, for each exploitable statement, the embedded monitor needs *t+c* unit additional time.

We consider Dell optiplex 980, Intel core i7 @ 2.80 GHz quad core machine as the evaluation environment. We execute database server and web server on the same machine. With this system specification, for each exploitable statement, the monitor introduces a maximum delay of 8 micro seconds.

Keizun *et al* [61] also use SchoolMate, WebChess, and FAQforge to evaluate their tool *ARDILLA* which can detect weak points of PHP/MySQL applications for SQLI attacks using taint analysis. Then the tool recognizes proper input set to exploit those weak points. They list the weak points for SQLI attack of these three applications in a website[1]. We consider the list as a benchmark to evaluate the SQLI monitor. We match the exploitable statements list with the weak points list and observe that our framework can detect all of the weak points successfully.

5.4 Summary

In this chapter, we have provided the implementation detail of DESERVE for SQLI monitoring. We have discussed inputs, outputs, and the source code processing techniques of the framework. The monitoring behavior of the embedded monitor has been described. We have measured the effectiveness of DESERVE by embedding

[1] http://groups.csail.mit.edu/pag/ardilla/

monitors on three open source programs. We have analyzed the performance of the framework and the embedded monitor and found it effective for SQLI monitoring.

Chapter 6

Cross-Site Scripting Monitoring

In this chapter, we discuss the framework implementation for XSS monitoring. The implementation details, inputs, and outputs of each step of the framework are presented in Section 6.1. We discuss the necessary parameters to design the runtime testing environment for the embedded monitor. Our framework DESERVE can embed monitor for both first and second order XSS attacks. We discuss the implementation details for both types of attack detection. The behavior of the XSS monitor to detect an attack is explained in Section 6.2. We evaluate the framework for three open source programs: SchoolMate_v1.5.4, WebChess v1.0.0rc2, and FAQforge-1.3.2. Finally, we summarize the performance of the framework and the embedded monitor in Section 6.3.

6.1 Monitor embedding framework

We employ DESERVE to embed monitor into PHP programs to detect XSS attacks. PHP is the most used server side scripting language and 81.9 percent (XSS, SQLI, path traversal etc.) of vulnerabilities in PHP applications are introduced due to the lack of input sanitization [59]. We discuss the three steps of the framework for XSS monitoring in Subsections 6.1.1-6.1.3.

6.1.1 Vulnerable statements marking

In this step, the monitor embedding framework marks all vulnerable statements. By vulnerable statements, we indicate those statements which can cause an XSS attack. Table 6.1 shows the functions responsible for an XSS attack in PHP. These two functions write their input parameters directly to a client browser. If any malicious JavaScript is used as input parameters for these functions, an XSS attack may occur. DESERVE takes the list of the vulnerable functions and PHP grammar as inputs in this step.

Table 6.1: Vulnerable functions in PHP responsible for XSS

Function	**Initial variable set V for slicing**
void echo (string $arg1 [, string $...])	$arg1, $arg2, …
int print (string $arg)	$arg

Figure 6.1 shows two code snippets from web applications WebChess and SchoolMate. After the first step, vulnerable statements are marked using XML like opening tag <*prblvulStmnt*> and closing tag </*prblvulStmnt*>.

Figure 6.2 shows the output after marking. The vulnerable statements at Line 3 of Figure 6.2(a) and Line 6 of Figure 6.2(b) are shown in grey background in this figure. TXL rules/functions insert these tags into proper places using PHP grammar written in TXL programming language.

```
1 $fromPerson=($_POST['from'])?$_POST['from']:"NULL";
2 $toPerson=($_POST['to'])?$_POST['to']:"NULL";
3 echo("From $fromPerson, To $toPerson<br>");
```

a. Snippet from sendmessage.php of WebChess

```
1 $query = mysql_query ("SELECT courseid, coursename FROM courses WHERE (teacherid =
$teacherid[0] OR substituteid = $teacherid[0]) AND semesterid = $_POST[semester]") or die
("ViewCourses.php: Unable to get a list of classes - ".mysql_error ());
2    $row = 0;
3   while ($class = mysql_fetch_row ($query))
4    {
5       $row ++;
6       print ("<tr class='".($row % 2 == 0 ? "even" : "odd")."'>
<td><a class='items'
href=\"JavaScript:document.classes.selectclass.value=$class[0];document.classes.page2.value=1;doc
ument.classes.submit();\" onMouseover=\"window.status='View Information For $class[1]';return
true;\" onMouseout=\"window.status='';return true;\">$class[1]</a></td>
</tr>");
}
```

b. Snippet from Viewcourses.php from SchoolMate

Figure 6.1: Example code snippet of XSS

6.1.2 Static backward slicing to identify exploitable vulnerable statements

DESERVE computes the slice of each marked vulnerable statement to identify exploitable statements for XSS attacks. By exploitable vulnerable statements, we mean those *echo* and *print* function calls whose executions are being influenced by a user input. Depending on the attack type, the user input can be either immediate (for first order attack) or stored (for second order attack).

Each marked statement serves as the program point of the slicing criteria. The other parameter of the slicing criteria (*V*) is determined from the parameters of the

vulnerable functions. Table 6.1 shows the initial value of the second parameter of the slicing criteria of vulnerable statements.

```
1 $fromPerson=($_POST['from'])?$_POST['from']:"NULL";
2 $toPerson=($_POST['to'])?$_POST['to']:"NULL";
3 <prblvulStmnt> echo("From $fromPerson, To $toPerson<br>");</prblvulStmnt>
```

a. Snippet from sendmessage.php of WebChess

```
1 $query = mysql_query ("SELECT courseid, coursename FROM courses WHERE (teacherid =
$teacherid[0] OR substituteid = $teacherid[0]) AND semesterid = $_POST[semester]") or die
("ViewCourses.php: Unable to get a list of classes - ".mysql_error ());
2    $row = 0;
3   while ($class = mysql_fetch_row ($query))
4    {
5       $row ++;
6       <prblvulStmnt> print ("<tr class='".($row % 2 == 0 ? "even" : "odd")."'>
<td><a class='items'
href=\"JavaScript:document.classes.selectclass.value=$class[0];document.classes.page2.value=1;docum
ent.classes.submit();\" onMouseover=\"window.status='View Information For $class[1]';return true;\"
onMouseout=\"window.status='';return true;\">$class[1]</a></td>
</tr>"); </ prblvulStmnt >
}
```

b. Snippet from Viewcourses.php from SchoolMate

Figure 6.2: Code snippet after vulnerable statements marking

A first order XSS attack occurs when an input from a user is used in an *echo* or *print* function. On the other hand, in a second order attack, an input stored in the database may cause the exploitation. As *$_POST*, *$_GET*, *$_COOKIE* and *$_REQUEST* contain user inputs, we consider a statement takes user input, if it uses any of these arrays. If the slice of a vulnerable statement uses any these arrays or parameters of the executing function, we consider that vulnerable statement exploitable by a first order attack. If the slice of a vulnerable statement has a database access function, we consider the vulnerable

statement exploitable. The list of PHP database access functions for accessing MySQL database server is provided in Table 5.2.

```
1 <mark>$fromPerson=($_POST['from'])?$_POST['from']:"NULL";</mark>
2 <mark>$toPerson=($_POST['to'])?$_POST['to']:"NULL";</mark>
3 <prblvulStmnt> echo("From $fromPerson, To $toPerson<br>"););</ prblvulStmnt >
```

Figure 6.3: Code snippet from sendmessage.php of WebChess after slicing

Figure 6.3 shows the slice of code snippet from sendmessage.php of WebChess (a web-based chess program) for a first order attack. The statements in a slice are enclosed within XML like opening tag <*mark*> and closing tag </*mark*>. The statement at Line 1, which is also in the slice, contains array *$_POST*. Therefore, the vulnerable statement at Line 3 may be exploited and our framework marks it as exploitable by changing the XML tag to <*vulStmnt*>.

Figure 6.4 shows the slices of the vulnerable statements of code snippet from Viewcourses.php of SchoolMate (a school management web application) for a second order attack. The statements in a slice are enclosed within XML like opening tag <*mark*> and closing tag </*mark*>. The vulnerable statement at Line 6 uses a variable *$class*. This variable is fetched from a database (Lines 1-3). Therefore, DESERVE marks it as exploitable by changing the XML tag to <*vulStmnt*>. Figure 6.5 shows the code snippet in Figure 6.1 after exploitable statement marking.

```
1 <mark>$query = mysql_query ("SELECT courseid, coursename FROM courses WHERE (teacherid
= $teacherid[0] OR substituteid = $teacherid[0]) AND semesterid = $_POST[semester]") or die
("ViewCourses.php: Unable to get a list of classes - ".mysql_error ());</mark>
2    <mark>$row = 0;</mark>
3   <mark>while ($class = mysql_fetch_row ($query))
4    {
5       <mark>$row ++;</mark>
6       <prblvulStmnt> print ("<tr class='".($row % 2 == 0 ? "even" : "odd")."'>
<td><a class='items'
href=\"JavaScript:document.classes.selectclass.value=$class[0];document.classes.page2.value=1;docu
ment.classes.submit();\" onMouseover=\"window.status='View Information For $class[1]';return true;\"
onMouseout=\"window.status='';return true;\">$class[1]</a></td>
</tr>"); </ prblvulStmnt >
}</mark>
```

Figure 6.4: Code snippet from Viewcourses.php from SchoolMate after slicing

6.1.3 Source code instrumentation

In this phase, DESERVE inserts code segment at exploitable vulnerable statements to make the application capable of self-monitoring. It needs to arrange a test environment to execute exploitable vulnerable statements. For XSS exploitation detection in PHP, we arrange a test inside a PHP program using a DOMDocument object. PHP DOMDocument object represents an entire HTML document. It provides functions and properties to access HTML tags individually.

The framework creates a DOMDocument object *$doc* using the input parameters of an exploitable statement. The framework creates another DOMDocument object *$docexpected* using sample expected values for those parameters. The expected input is needed to be entered into the framework. Let us consider that the set of HTML tags in *$doc* is *S* and in *$docexpected* is *E*. The number of elements in *S* and *E* are $|S|$ and $|E|$, respectively. After creating two DOM objects, we check the following conditions: $|S| =$

$|E|$ and $S = E$. If any of these conditions fails, the echo/print function is not safe and the function is not executed.

```
1 $fromPerson=($_POST['from'])?$_POST['from']:"NULL";
2 $toPerson=($_POST['to'])?$_POST['to']:"NULL";
3 <vulStmnt> echo("From $fromPerson, To $toPerson<br>"););</ vulStmnt >
```

a. Snippet from sendmessage.php of WebChess

```
1 $query = mysql_query ("SELECT courseid, coursename FROM courses WHERE (teacherid =
$teacherid[0] OR substituteid = $teacherid[0]) AND semesterid = $_POST[semester]") or die
("ViewCourses.php: Unable to get a list of classes - ".mysql_error ());
2    $row = 0;
3   while ($class = mysql_fetch_row ($query))
4    {
5       $row ++;
6       <vulStmnt> print ("<tr class='".($row % 2 == 0 ? "even" : "odd")."'>
<td><a class='items'
href=\"JavaScript:document.classes.selectclass.value=$class[0];document.classes.page2.value=1;docu
ment.classes.submit();\" onMouseover=\"window.status='View Information For $class[1]';return true;\"
onMouseout=\"window.status='';return true;\">$class[1]</a></td>
</tr>"); </ vulStmnt >
}
```

b. Snippet from Viewcourses.php from SchoolMate

Figure 6.5: Code snippet after exploitable statements marking

Figure 6.6 shows the modified code snippet of Figure 6.1(a) after the code instrumentation. The parameter of function *echo* is assigned into variable *$htmltext* at Line 3. DOMDocuments *$doc* and *$docsexpected* load the *$htmltext* and safe input *$htmltextexpected*, respectively. The number of HTML tags is compared in Line 10. Each tag is compared individually at Lines 12-16. If both objects have exactly the same set of HTML tags, the echo function is safe for the execution (Line 18).

```
1 $fromPerson=($_POST['from'])?$_POST['from']:"NULL";
2 $toPerson=($_POST['to'])?$_POST['to']:"NULL";
3 $htmltext ="From $fromPerson, To $toPerson<br>";
4 $doc = new DOMDocument();
5 $docexpected = new DOMDocument();
6 $doc->loadHTML($htmltext);
7 $docexpected->loadHTML($htmltextexpected);
8 $items = $doc->getElementsByTagName('*');
9 $itemsexpected = $doc1->getElementsByTagName('*');
10 if ($items->length==$itemsexpected->length){
11 $i=0;
12 for ($i = 0; $i < $items->length; $i++) {
13 if($items->item($i)->nodeName!=$itemsexpected->item($i)->nodeName){
14       break;
15    }
16 }
17 if ($i==$items->length){
18 echo $htmltext;
19 }
```

Figure 6.6: Code snippet from sendmessage.php of WebChess after instrumentation

6.2 Monitoring

Figure 6.7 shows the XSS monitoring flowchart of the embedded monitor. Whenever any program execution reaches up to an exploitable vulnerable statement, a DOMDocument object is created using the parameters of the function. At Step 2 of this flowchart, the DOMDocument object is compared with a DOMDocument object created from the sample expected input. The exploitation is detected using the conditions described in Subsection 6.1.3. If there is no exploitation, the exploitable statement is executed.

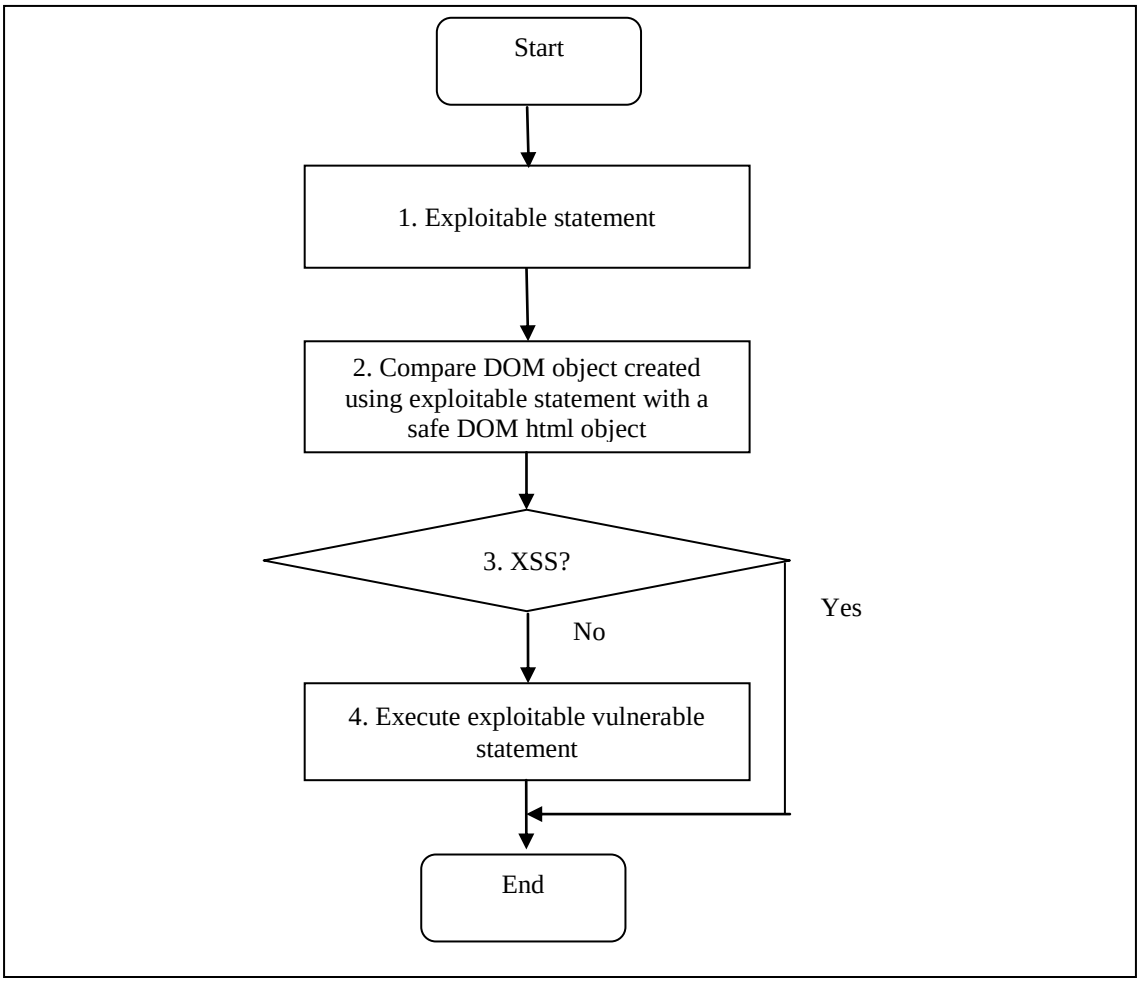

Figure 6.7: XSS monitoring

6.3 Evaluation

We apply DESERVE to embed XSS monitors in SchoolMate_v1.5.4, WebChess v1.0.0rc2, and FAQforge-1.3.2 [68-70]. The first step of DESERVE marks vulnerable statements. We recognize 285 vulnerable statements in SchoolMate from the outcome of the first step. WebChess and FAQforge have 436 and 88 vulnerable statements, respectively.

The second step of the framework marks exploitable statements. For first order attacks, the number of exploitable statements in SchoolMate, WebChess, and FAQforge are 60, 28, and 20, respectively. The number of exploitable vulnerable statements is 6.4-23 percent of the number of vulnerable statements. The individual break down for each vulnerable statement pattern is presented in Table 6.2.

The static backward slicing at the second step of DESERVE identifies only 95 vulnerable statements in SchoolMate which are exploitable for the second order attacks. WebChess has 36 exploitable statements. FAQforge does not have any exploitable statement for a second order XSS attack. The number of exploitable vulnerable statements is 0-33 percent of the number of vulnerable statements. The detail statistics is presented in Table 6.3.

We calculate the static and dynamic overheads introduced in our monitoring approach. We instrument the source code at exploitable vulnerable statements. The instrumentation increases the line of code. Table 6.4 and Table 6.5 show the data about the additional line of code after the instrumentation for the first and second order attacks, respectively. For the first order attacks, the additional line of code instrumentation is 10-25 percent of the total line of code. The line of code is increased due to the instrumentation by 0-19 percent for the second order attacks.

As we employ runtime testing to detect an exploitation, the monitor introduces some delay before each exploitable vulnerable statement execution. For monitoring each exploitable statement, we need two DOMDocument objects creation overhead. Also, our

approach introduces overhead for comparing two DOMDocument objects. We consider Dell optiplex 980, intel core i7 @ 2.80 GHz quad core machine as the evaluation environment. The monitor introduces 0.5 micro second delays to compare the largest DOMDocument objects from SchoolMate, WebChess, and FAQforge.

Table 6.2: Exploitable statements in SchoolMate, WebChess, and FAQforge for first order XSS attack

Applications	**SchoolMate**			**WebChess**			**FAQforge**		
Vulnerable functions	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement
Echo	1	0	0	436	28	6.4	16	0	0
Printf	284	60	21.1	0	0	0	72	20	28
Total	**285**	**60**	**21**	**436**	**28**	**6.4**	**88**	**20**	**23**

Table 6.3: Exploitable statements in SchoolMate, WebChess, and FAQforge for second order XSS attack

Applications	**SchoolMate**			**WebChess**			**FAQforge**		
Vulnerable functions	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement	#vulnerable statement	#exploitable statement	% of exploitable statement
Echo	1	0	0	436	36	8.25	16	0	0
Print	284	95	33	0	0	0	72	0	0
Total	**285**	**95**	**33**	**436**	**36**	**8.25**	**88**	**0**	**0**

.

Keizun *et al* [61] propose a tool to identify weak points of a PHP program for XSS attacks. Later they apply fuzz testing to exploit the weak points. They apply the tool on SchoolMate, WebChess, and FAQforge. They list the weak points of these web

applications in a website[2]. We match the exploitable statement list with the weak point list and observe that our framework detects most of the weak points successfully. Very few weak points (four exploitable statements for the first order XSS attacks and one exploitable statement for the second order XSS attack for SchoolMate) are not detected as a global variable is involved in the slice. DESERVE cannot determine a slice across multiple files.

Table 6.4: Added line of code after code instrumentation for first order XSS attack

Applications	**SchoolMate**	**WebChess**	**FAQforge**
Line of Code	6,554	3,640	1,045
Line of code added	780	364	260
% increased	12	10	25

Table 6.5: Added line of code after code instrumentation for second order XSS attack

Applications	**SchoolMate**	**WebChess**	**FAQforge**
Line of Code	6,554	3,640	1,045
Line of code added	1235	468	0
% increased	19	13	0

6.4 Summary

We have discussed the implementation details of each of the steps of the framework for XSS attack detection. We have explained the inputs, outputs, and implementation of the framework with real world program examples. The slicing, the runtime testing, and the source code instrumentation have been discussed. We have

[2] http://groups.csail.mit.edu/pag/ardilla/

described the behavior of the monitor for detecting XSS exploitations. We have employed this framework to embed XSS monitors into three well known open source programs. Finally, the analysis of the result has been presented.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We propose a monitor embedding framework DESERVE to combine the benefit of static analysis and testing. Our monitor embedding framework and monitoring technique use static analysis (static backward slicing) and dynamic analysis (runtime testing) to achieve both completeness and soundness. Our approach can monitor effectively with negligible overhead.

We apply this framework for BOF, SQLI, and XSS which have been appearing in "Top 25 most dangerous software error" list consistently [1]. DESERVE identifies the number of vulnerable statements that are exploitable. Later, the monitor can test any vulnerability exploitation successfully. We apply DESERVE for three types of vulnerabilities and different applications.

We define DESERVE in a way such that it can be easily extendable for new languages and for new vulnerabilities. DESERVE can embed monitors to detect BOF, SQLI, and XSS vulnerabilities of applications implemented in C/C++ and PHP. The monitors can successfully detect some attacks. The XSS and SQLI monitors can effectively detect both first and second order code injection attacks. The framework identifies the exploitable statements from source code and embeds a monitor into the

source code. The embedded monitor detects an attack using runtime testing. It executes the exploitable statement in a separate process.

Our work avoids the program state saving and information extraction overheads. Traditional monitors examine each change of certain properties of a program to detect an attack. However; each change does not initiate an attack. We applied static backward slicing to identify the potential changes. Some monitors track the user input flow to find out the potential attack change. This approach introduces runtime overhead. A traditional monitor needs state comparison to detect any attack, which may not always detect an attack before the exploitation. Therefore, we test the potential vulnerable statement in a separate environment to test the exploitable statement at runtime.

7.2 Limitation and future work

In program slicing, we include a statement to a slice, if the modified expression has any variable that is used in any of the statements in the current slice. However, all the variables in a modifying expression may not necessarily change the modified variables. It depends on the application language and the implementation. In future, we will improve our slicing so that it can determine a slice more precisely. Using static slicing, DESERVE cannot handle data flow across multiple source files that use global variables. As a result, it may introduce false negatives. We plan to employ an advanced slicing approach to resolve this issue.

To find out exploitable vulnerable statements for second order code injection, we assume that any input from the database which is used in vulnerable statement is

dangerous. This assumption can lead to some false positives. All data tables in a database may not contain user data. A database may have tables to manage system settings and configurations. We plan to introduce a flag to differentiate between the data entered by users and the system data. We will change the slicing so that it can consider only user data.

DESERVE embeds a monitor automatically. However, due to language constraints, sometimes manual intervention is necessary. . For example, C/C++ language supports pre-processor directives. These directives are processed at pre-processing step before compilation. The syntactical structure of the source code may not obey the C/C++ grammar before pre-processing because of the pre-processor directives. This phenomenon sometimes makes automatic slicing impossible for DESERVE. This problem can be solved by preprocessing before slicing. However, depending on the system setup, often the pre-processor discards source code. Therefore, slicing on the preprocessed source code may not identify all exploitable statements. Currently, we correct the syntactical structure of the source code manually, if necessary. However, we plan to improve our framework implementation so that it can do the syntactical change automatically. Finally, we want to expand the framework implementation for the vulnerabilities in other implementation languages.

References

[1] CWE - 2010 CWE/SANS Top 25 Most Dangerous Software Errors, http://cwe.mitre.org/top25/ (Accessed in June, 2011).

[2] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," *Proc. of the Network and Distributed System Security Symposium,* California, USA, February 2000, pp. 3–17.

[3] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computer & Operation Research, Elsevier,* vol. 35, Issue 10, October 2008, pp. 3125-3143.

[4] M. Weber, V. Shah, and C. Ren, "A case study in detecting software security vulnerabilities using constraint optimization," *Proc. of the 1st Intl. Workshop on Source Code Analysis and Manipulation,* Florence, Italy, November 2001, pp. 1-11.

[5] ITS4: Software Security Tool [Cigital], http://www.cigital.com/its4/ (Accessed in June, 2011).

[6] Flawfinder Home Page, http://www.dwheeler.com/flawfinder/ (Accessed in June, 2011).

[7] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," *Proc. of the International Symposium on Software Testing and Analysis,* London, United Kingdom, July 2007, pp. 196-206.

[8] S. Gupta, P. Pratap, H. Saran, and S. Arun-Kumar, "Dynamic code instrumentation to detect and recover from return address corruption," *Proc. of the International Workshop on Dynamic Systems Analysis*, Shanghai, China, May 2006, pp. 65-72.

[9] L. C. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," *Proc. of the 22nd Annual Computer Security Applications Conference*, Miami Beach, FL, December 2006, pp. 463-472.

[10] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," *Proc. of the Network and Distributed System Security Symposium*, San Diego, California, USA, February 2005.

[11] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu, "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)," *Proc. of the 20th Annual Computer Security Applications Conference*, Tucson, AZ, December 2004, pp. 82-90.

[12] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, Tucson, AZ, USA, June 2008, pp. 206-215 .

[13] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," *Proc. of the Network and Distributed System Security Symposium,* San Diego, CA, February 2008.

[14] H. Shahriar and M. Zulkernine, "Taxonomy and Classification of Automatic Monitoring of Program Security Vulnerability Exploitations," *Journal of Systems and Software (JSS)*, Elsevier, Vol. 84, Issue 2, February 2011, pp. 250-269.

[15] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," *Proc. of IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, Arlington, Virginia, March 2006.

[16] Web Application Exploits and Defenses, http://google-gruyere.appspot.com/ (Accessed in June, 2011).

[17] L. C. Ng, "Dependence Graphs and Program Slicing," http://www.grammatech.com/ research/papers/slicing/slicingWhitepaper.html.

[18] C. Murphy, G. E. Kaiser, and M. Chu, The In Vivo Approach to Testing Software Applications, Technical Report (cucs-007-08), Department of Computer Science, Columbia University, USA, 2008

[19] H. Dai, C. Murphy, and G. Kaiser, "Configuration fuzzing for software vulnerability detection," *Proc. of the International Conference on Availability, Reliability and Security*, Krakow, Poland, February 2010, pp. 525-530

[20] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," *Proc. of the 34th Annual Intl. Symposium on Computer Architecture*, California, June 2007, pp. 482-493..

[21] M. W. Stephenson, R. Rangan, E. Yashchin, and E. Van Hensbergen, "Statistically regulating program behavior via mainstream computing," *Proc. of the 8th Annual*

IEEE/ACM International Symposium on Code Generation and Optimization, San Jose, California, March 2010, pp. 238-247.

[22] T. Chiueh and F. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," *Proc. of the Intl. Conf. on Distributed Computing Systems*, Arizona, April 2001, pp. 409-417.

[23] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks," *Proc. of the 7th USENIX Security Conference*, San Antonio, Texas, January 1998.

[24] A. Aggarwal and P. Jalote, "Monitoring the security health of software systems," *Proc. of the 17th International Symposium on Software Reliability Engineering*, North Carolina, November 2006, pp. 146-158.

[25] B. Madan, S. Phoha, and K. Trivedi, "StackOFFence: A Technique for Defending Against Buffer Overflow Attacks," *Proc. of the Intl. Conference on Information Technology: Coding and Computing*, Las Vegas, April 2005, pp. 656- 661.

[26] M. Prasad and T. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," *Proc. of the USENIX Annual Technical Conference,* San Antonio, Texas, June 2003, pp. 211–224.

[27] ChangwooPyo, ByungchulBae, Taejin Kim, and Gyungho Lee, "Run-time detection of buffer overflow attacks without explicit sensor data objects," *Proc. of the Information Technology: Coding and Computing*, Las Vegas, Nevada, April 2004, pp. 50-54.

[28] C. Fetzer and Z. Xiao, "Detecting heap smashing attacks through fault containment wrappers," *Proc. of the 20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, USA, October 2001, pp. 80-89.

[29] R. Jones and P. Kelly, “Backwards -compatible Bounds Checking for Arrays and Pointers in C Programs,” *Proc. of Automated and Algorithmic Debugging*, Sweden, 1997, pp. 13-26.

[30] R. Hastings and B. Joyce, “Purify: Fast Detection of Memory Leaks and Access Errors,” *Proc. of the USENIX Winter Conference*, San Francisco, January 1992, pp. 125-138.

[31] K. Lhee and S. J. Chapin, “Type-Assisted Dynamic Buffer Overflow Detection,” *Proc. of the 11th USENIX Security Symposium*, San Francisco, August 2002, pp. 81-90.

[32] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," *Proc. of the International Conference on Dependable Systems and Networks*, Philadelphia, June 2006, pp. 269-280.

[33] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," Proc. of the 11th Annual Network and Distributed System Security Symposium, San Diego, California, February 2004, pp. 159–169.

[34] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," *Proc. of the 18th Conference on USENIX Security Symposium*, California, USA, August 2009, pp. 51-66.

[35] H. Han, X. L. Lu, L. Y. Ren, B. Chen, and N. Yang, "AIFD: A runtime solution to buffer overflow attack," *Proc. of the International Conference on Machine Learning and Cybernetics*, Hong Kong, August 2007, pp. 3189-3194.

[36] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," *Proc. of the 11th USENIX Security Symposium*, San Francisco, August 2002, pp. 191-206.

[37] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "Accmon: Automatically detecting memory-related bugs via program counter-based invariants," *Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Portland, Oregon, December 2004, pp. 269-280

[38] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," *Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006, pp. 158-168.

[39] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a board range of memory error exploits," *Proc. of the 12th Conference on USENIX Security Symposium*, Washington DC, August 2003, pp. 8-8.

[40] PaxProject, http://pax.grsecurity.net/docs/pax.txt (Accessed in June, 2011) .

[41] H. Etoh, GCC Extension for Protecting Applications from Stack-smashing Attacks, http://www.trl.ibm.com/projects/ security/ssp.

[42] Y. Younan, W. Joosen, and F. Piessens, "Efficient protection against heap-based buffer overflows without resorting to magic," *Proc. of* the *Information and Communications Security,* Raleigh, USA, December 2006, pp. 379-398.

[43] Y. Younan, F. Piessens, and W. Joosen, "Protecting global and static variables from buffer overflow attacks," *Proc. of the International Conference on Availability, Reliability and Security,* Fukuoka, Japan, March 2009, pp. 798-803.

[44] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," *Proc. of the 10th ACM Conference on Computer and Communications Security,* Chicago, October 2003, pp. 272-280.

[45] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan, "CANDID: Preventing sql injection attacks using dynamic candidate evaluations," *Proc. of the 14th ACM Conference on Computer and Communications Security,* Virginia, October 2007, pp. 12-24.

[46] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," *Proc. of the 5th International Workshop on Software Engineering and Middleware,* Lisbon, Portugal, September 2005, pp. 106-113.

[47] A. Alfantookh, "An automated universal server level solution for SQL injection security flaw," *Proc. of the 2004 International Conference on Electrical, Electronic and Computer Engineering (ICEEC'04),* Cairo, Egypt, September 2008, pp. 131-135.

[49] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," Proc. of the *Applied Cryptography and Network Security,* Yellow Mountain, China, June 2004, pp. 292-302.

[49] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-side detection of cross-site scripting attacks," Proc. of the *Annual Computer Security Applications Conference,* Anaheim, California, December 2008, pp. 335-344.

[50] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks," *Detection of Intrusions and Malware, and Vulnerability Assessment,* Paris, France, July 2008, pp.23-43.

[51] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: Mitigating XSS attacks using a reverse proxy," Proc. of the ICSE Workshop on *Software Engineering for Secure Systems,* Vancouver, May 2009, pp. 33-39.

[52] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-based spyware detection," *Proc. of the 15th conference on USENIX Security Symposium,* Vancouver, July 2006.

[53] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," *Proc. of the 16th International Conference on World Wide Web,* Alberta, May 2007, pp. 601-610.

[54] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A proposal and implementation of automatic detection/collection system for cross-site scripting

vulnerability," *Proc. of the 18th International Conference on Advanced Information Networking and Applications,* Fukuoka, Japan, March 2004, pp. 145-151.

[55] G. Iha and H. Doi, "An implementation of the binding mechanism in the web browser for preventing XSS attacks: Introducing the bind-value headers," *Proc. of the International Conference on Availability, Reliability and Security,* Fukuoka, Japan, March 2009, pp. 966-971

[56] TXL Home Page, http://www.txl.ca/ (Accessed in June, 2011).

[57] G. Ollmann, "Second-order code injection attacks," *NTGS Software Insight Security Research,* 2004..

[58] Usage Statistics and Market Share of Server-side Programming Languages for Websites, http://w3techs.com/technologies/overview/programming_language/all (Accessed in June, 2011).

[59] PHP-related vulnerabilities on the National Vulnerability Database, http://www.coelho.net/php_cve.html (Accessed in June, 2011) .

[60] MySQL Official Website, http://www.mysql.com/ (Accessed in June, 2011).

[61] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," *Proc. of the 31st International Conference on Software Engineering,* Vancouver, May 2009, pp. 199-209.

[62] R. Richardson, Computer Crime and Security Survey, Computer Security Institute, VIII(1):1-21, 2003. www. gocsi.com/press/20030528.html.

[63] CVE - Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org/ (Accessed in June, 2011).

[64] Python Programming Language, http://www.python.org/ (Accessed in June, 2011).

[65] WU-FTPD, http://en.wikipedia.org/wiki/WU-FTPD (Accessed in June, 2011).

[66] The gzip home page, http://www.gzip.org/ (Accessed in June, 2011).

[67] Welcome to the Crafty Chess page, http://www.craftychess.com/ (Accessed in June, 2011).

[68] SchoolMate, http://sourceforge.net/projects/schoolmate/ (Accessed in June, 2011).

[69] Welcome to WebChess, http://sourceforge.net/apps/mediawiki/webchess/index.php?title=Main_Page (Accessed in June, 2011).

[70] FAQForge, http://sourceforge.net/projects/faqforge/ (Accessed in June, 2011).

[71] P. Bathia, B. R. Beerelli, and M. Laverdi`ere, and M. D. Ernst, "Assisting Programmers Resolving Vulnerabilities in Java Web Applications," *Proc. of the 1st international Conference on Computer Science and Information Technology,* Bangalore, India, January 2011, pp. 268-279.