Buffer Overflow Vulnerability Detection based on Format-Matching on Source Level

Xiaoyu Wang
State Key Laboratory of Network and Switching Technology
Beijing University of Posts and Telecommunications
Beijing, 100876, China
aphay@bupt.edu.cn

Zhao Zhang
State Key Laboratory of Network and Switching Technology
Beijing University of Posts and Telecommunications
Beijing, 100876, China

Qiaoyan Wen
State Key Laboratory of Network and Switching Technology
Beijing University of Posts and Telecommunications
Beijing, 100876, China

Abstract—Buffer overflow has become the most common software vulnerability, which seriously restricts the development of the software industry. It's very essential t o find out an effective method to detect this kind of software bugs accurately. In this paper, we design an improved buffer overflow detection system. At first, our system preprocesses the source code to add some auxiliary detection symbols. Then, it scans the source code by a static detector, which uses the identifier for auxiliary detection and combines with a dynamic detection method to improve the recognition accuracy and detection capability. Finally, we make a comparison between our system and the original detection system. To assess the usefulness of this approach, several experiments are performed on a simulation system, and we can draw a conclusion that our system performs better than other detection software. The method proposed in this paper is of the important application value and can improve detection accuracy.

Keywords-buffer overflow, rule-based detection, dynamic test, format-matching

I. INTRODUCTION

Buffer overflow (BOF) is source code level vulnerability that allows unchecked inputs to overflow data buffers during memory writing operations. If inputs are crafted carefully, shell code can be provided as the content for copy operations. This may lead to not only abnormal program behaviors, but also unwanted code execution. Even system root authority may be obtained by unexpected hackers. The presence of BOF opens the door for many notorious types of attacks such as injection of malicious worms. Recent vulnerability reports suggest that BOF vulnerabilities still exist in both legacy and newly developed applications and cause many safety problems of user data. Thus, detecting BOF vulnerabilities from programs is important.

There are already a lot of researches about buffer overflow detection. The detection technologies for this vulnerability are divided into two categories[1][2]: static detection and dynamic detection. Static detection is efficient, but not accurate. It has a high rate of mistake. On the other hand, dynamic test can search vulnerabilities more accurately than static detection, but it runs slowly and inefficiently. Buffer overflow detection has been studied extensively. Most of these works is a discussion of static methods. Bernhard J. Berger and Karsten Sohr[1] proposed a detection method. They use both Soot and Bauhaus to make an analysis of reverse engineering. Chen Liu, and Jinqiu Yan[2] propose an analysis approach, R2Fix, for bug reports to automatically generate bug-fixing patches. Except these works, we find out that rule-based source level patching method proposed by Hossain Shahriar and Hisham M. Haddad[3], performances well in most instances. Wei Le[4] also proposed a dynamic detection structure in one of his recently papers, which is easily combined with static methods.

This paper proposes an improved detection system. In summary, this work makes the contributions as follows:

- We design a pretreatment module, which can help static scanners search the source code very easily.
- We expand the Recognizer of the static part. Rule-based method is based on format-matching[3] in Recognizer. Some new code patterns are added into our system.
- We combine static method with a dynamic module. This structure may improve the static detection because of the accuracy of dynamic test.
- We implement the system by Python, a kind of easy-expand script language. It makes our system easy to work with other software.
- We make an experiment to test our detection system's performance. The result proves that our

approach can find buffer overflow vulnerabilities more accurately.

The rest of this paper is organized as follows. We briefly describe the basic theory of our method in section II, and in section III we present the design of our system. We demonstrate the experimental results and evaluate our approach in section IV. In the end, we summarized this paper in section V.

II. THEORY

Before designing the detection system, we introduce some basic theories.

Buffer overflow vulnerabilities are caused by programming errors, which allows an attacker to cause the program to write beyond the bounds of an allocated memory block to corrupt other data structures[8].

Common sources of BOF (shortly for Buffer Overflow) vulnerabilities include unsafe library function calls, buffer index variables, absence of null characters, arithmetic operations using pointers, and pointer usage in complex code blocks such as loop and if structures[3]. Rule-based source level detection includes identifying programming elements that might cause BOF, such as limitations due to languages, associated libraries, and logical errors. It is based on match several code patterns, including simple (one statement) and complex (multiple statements) forms of BOF. We can extend additional code patterns to make the detection more accurately.

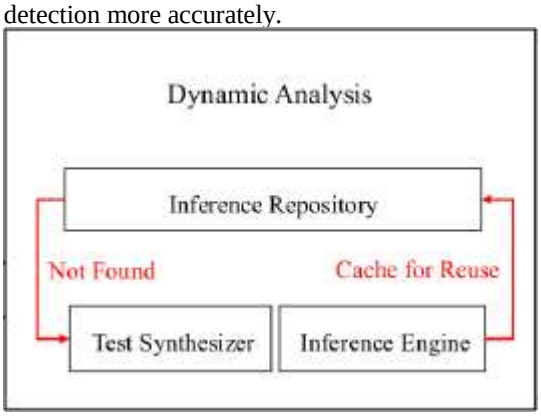

Figure 1. Dynamic Module

Based on the rule-based detection approach, we add a dynamic module to verify static test results.

The dynamic module is firstly proposed by Wei Le in Segmented Symbolic Analysis Research[4]. The dynamic analysis, illustrated in figure. 1, consists of three parts: an inference repository, a test synthesizer and an inference engine.

III. DESIGN OF OUR SYSTEM

We design a better detection system based on the previous research foundation. This detection system, illustrated in figure.2, combines the static method and the dynamic testing, which is a block diagram and reflects our vision and design for this detection system. We design our system into two parts: static part and dynamic part, on behalf of the static detection and dynamic detection. In the below diagram, the blue line represents the control flow among the components, and the green thick line indicates the data flow.

A. *Static search*

The input data is a piece of C/C++ program source code. At first, we do some pretreatments for the input data, such as adding some auxiliary detected symbols and other factors. At this step, we scan the whole input data to find out unsafe functions and other code segments which may cause buffer overflow. Detection system annotates the searched code segments using special comments, which includes the information about buffer type, definition, and length. These comments are written in a particular format, which is easy for the program to read the information. An example in Table I shows how to annotate the code.

Then, the program scans the code with its static scanner, while the recognizer module assists in the analysis to improve the accuracy of recognition and detection. In this process, the scanner extracts code information from the comments added by pretreatment module. Based on this information, the scanner can extract the relevant code fragments and put them together to build an independent test unit.

TABLE I. PRETREATMENT FOR SOURCE CODE

Before pretreatment	After pretreatment

char buf[1024];
…
strcpy(buf,dest);
…
char dest[65600];
for (i=0; i < sizeof (src); i++){
if(i<sizeof(dest)&&i>=0)
dest[i] = src[i];
}
	char buf[1024];/*@def id:1 len:1024*/
	…
	strcpy(buf,dest);/*@func id:1*/
	…
	char dest[32];/*def id:2 len:32*/
	for (i=0; i < sizeof (src); i++){
	if(i<sizeof(dest)&&i>=0)/*@check id:2*/
	dest[i] = src[i];/*@write id:2*/
	}

The recognizer module is the core of our static part. Some typical code patterns are stored in this module, including four simple patterns and one complex patterns: unsafe function call, illegal index, no '\0', illegal pointer position, and no pointer check inside loop block (or if block). Except these, we can also add code patterns to the recognizer according to the latest researches. The recognizer checks whether this extracted piece of code contains buffer overflow vulnerabilities, and returns a detection result back to the scanner. The scanner determines whether the code be passed to the dynamic part. After the static scanning, intermediate data with specific format is generated.

B. *Dynamic test*

After that, the intermediate data is submitted to the dynamic part along with the source code. The test synthesizer is responsible for constructing a dynamic test suite, and passing the test suite to inference engine for dynamic testing. This module seems to be a compiler or a unit test program, but it automatically builds a set of executable tests based on the information in the intermediate data.

In order to monitor the running status, we need an independent module. Inference engine is designed to determine whether the unit test triggers a memory error. It performs a regression analysis on the inputs and outputs of the tests and returns the discovered transfer functions and symbolic values. The inference repository stores all the inferred results for reuse. After all testing, we make a testing report to show the detection results. The test results

stored in the repository are important reference when the detection system constructs the final output.

C. Process Control

The entire program contains many modules, and is a complicated large system. It needs a global module to control the program running state and ensure that the system can detect buffer overflow vulnerabilities step by step. Thus, we design a controller and a state machine. The working process of control modules is as follow:

- When a module starts to work, it passes a message to the controller to report the state.
- The controller receives the message, and passes the message to the state machine.
- The state machine acknowledges the current operational status based on the messages and determines the next state of the module.

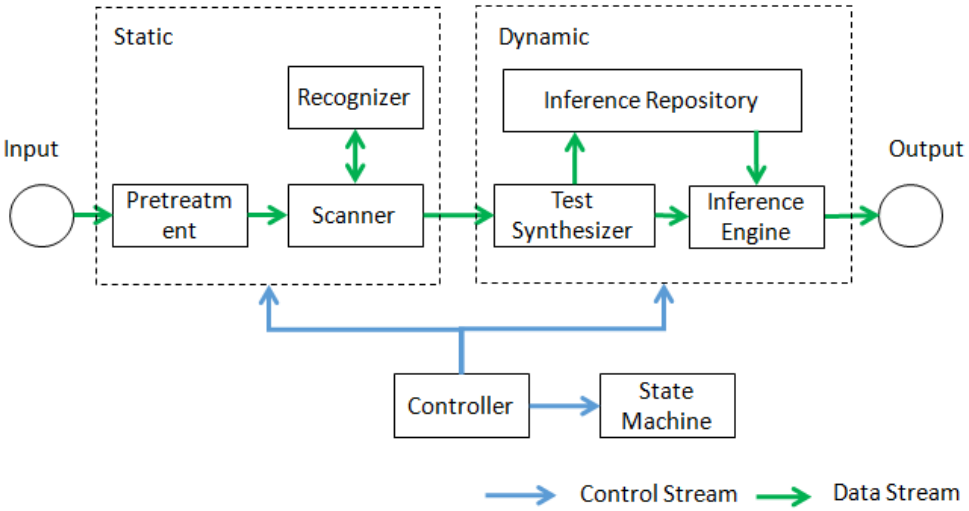

Figure 2. System Block Diagram

- The state result is returned back to Controller. And Controller sends a command to the module.
- Then the module can acknowledge his next job.

By the Controller, we are able to control all the modules and view their working state easily.

IV. Experiment and Result

Here is an example to illustrate how the system detects a piece of C++ program source code. The code in figure. 3 is a typical case which may cause buffer overflow. We pass this piece of code to our system and the steps of detection are as follows:

- The system adds a symbol after the function declaration of copyout, and the position, where the parameters are defined, is also marked.
- The scanner picks up the relevant code block very quickly with the help of those symbols inserted before.
- The system matches the function copyout with the patterns and judges whether it has caused troubles.
- The dangerous code block of copyout will be passed to the dynamic part.

```
#include "stdio.h"
#include "string.h"
void copyout(const char *input)
{
        char buf[10];
        strcpy(buf, input);
        printf("%s \n", buf);
};

void bar(void)
{
        printf("being hacked\n");
}

int main(int argc, char *argv[])
{
        copyout(argv[1]);
        return 0;
}
```

Figure 3. A piece of code with buffer overflow

- The system will compile the code under testing and the inference engine will execute the test.
- If a memory error happens, the system will report and print a piece of information to output.

In order to evaluate the effectiveness of our system, we make some experiments. There are three goals in our experiments: 1) to determine the capabilities of our analysis in handling loops and library calls and detecting bugs and infeasible paths;2) the capabilities of regression based dynamic inference in discovering correct transfer functions; and 3) the scalability of concurrent, on-demand hybrid analysis.

In the experiments, we compare the behavior of several different programs: Fority, ITS4 and BOON. We calculate the detection time, and keep an account of the detection result in Table II.

TABLE II. EXPERIMENT RESULT

Tested Software	**Consequence (compared with our system)**		
	bugs	***time***	***mistake rate***
Fority	more	shorter	a little lower
ITS4	more	shorter	lower
BOON	less	much shorter	lower

With the analysis of the results, we find out that the accuracy is increased about 10%, and the detection effect is almost not influenced. From the Table II, we can conclude that our system search more bugs than the Fority. However, compared with the ITS4 and the BOON, the search ability of our system is not much stronger. Although the detection time is not generally much improved, our system perfected better than the BOON for the detection speed.

V. CONCLUSION

This paper presents format-matching analysis, and demonstrates a detection system that flexibly weaves static and dynamic analyses on demands for their maximum capabilities of discovering program semantic information. In this system, we apply a rule-based approach in the static part, and combine the algorithm with a dynamic part of a unit test generation system. We implement our idea, and make several experiments to show that our system can address the loops and library calls, which cannot be analyzed by traditional analysis. It is fully automatic and can be generally applied for determining different program properties and for different programs.

The future goal is to formulate code rules for other BOF vulnerabilities, including loop and if statement expressions used in library and user-defined function calls and go to structures. Besides, future work also includes designing more effective dynamic test module and recursive calls and to further improves the code partition strategies.

ACKNOWLEDGMENT

This work has been supported by the State Key Laboratory of Network and Switching Technology, Beijing University of Posts and Telecommunications, and especially has been supported by National Natural Science Foundation of China (Grant Nos. 61300181, 61272057, 61202434, 61170270, 61100203, 61121061).

REFERENCES

[1] Bernhard J. Berger, Karsten Sohr, Rainer Koschke. “Extracting and Analyzing the Implemented Security Architecture of Business Applications,” 17th European Conference on Software Maintenance and Reengineering, 2013, 285-294

[2] Chen Liu, Jinqiu Yang, Lin Tan. “R2Fix: Automatically Generating Bug Fixes from Bug Reports,” IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, 282-291

[3] Hossain Shahriar, Hisham M. Haddad. “Rule-Based Source Level Patching of Buffer Overflow Vulnerabilities,” 10th International Conference on Information Technology: New Generations, 2013, 627-632

[4] Wei Le. “Segmented Symbolic Analysis” the International Conference on Software Engineering, 2013, 212-221

[5] Gabriel Scalosub, Peter Marbach, Jörg Liebeherr. “Buffer Management for Aggregated Streaming Data with Packet Dependencies,” IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 24, NO. 3, 2013, 439-449

[6] Andrzej Bartoszewicz, Piotr Lesniewski. “Variable Structure Flow Controller for Connection-Oriented Communication Networks,” 14th International Carpathian Control Conference, 2013, 5-10

[7] Krishna Jagannathan, Eytan Modiano. “The Impact of Queue Length Information on Buffer Overflow in Parallel Queues,” IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 59, NO. 10, 2013, 6393-6404

[8] Ping-Chen Lin, Ray-Guang Cheng, Yu-Jen Chang. "A Dynamic Flow Control Algorithm for LTE-Advanced Relay Networks," Vehicular Technology, VOL. 63, 2013, 334-343

[9] Ognenoski O., Martini M.G., Amon P.. "Segment-based teletraffic model for MPEG-DASH," Multimedia Signal Processing (MMSP), 2013, 333-337

[10] Zhiyuan An, Liu Haiyan. "Locating the Address of Local Variables to Achieve a Buffer Overflow," Computational and Information Sciences (ICCIS), 2013, 1999-2002

[11] Seon-Ho Park, Young-Ju Han, Soon-jwa Hong. "The Dynamic Buffer Overflow Detection and Prevent ion Tool for Windows Executables Using Binary Rewriting," Advanced Communication Technology, VOL 3, 2007, 1776-1781