Research on Buffer Overflow Test Based on Invariant

Fanping Zeng
Department of Computer, University of Science and Technology of China
Anhui Province Key Lab of Software in Computer and Communication
Hefei, Anhui, PR China
email:billzeng@ustc.edu.cn

Minghui Chen
Department of Computer, University of Science and Technology of China
Hefei, Anhui, PR China
email:mhchen@mail.ustc.edu.cn

Kaitao Yin
Department of Computer, University of Science and Technology of China
Hefei, Anhui, PR China
email:yinkt@mail.ustc.edu.cn

Xufa Wang
Department of Computer, University of Science and Technology of China
Anhui Province Key Lab of Software in Computer and Communication
Hefei, Anhui, PR China
email:xfwang@ustc.edu.cn

Abstract—**Buffer overflow (BOF) is one of the major vulnerabilities that lead to non-secure software. Testing an implementation for BOF vulnerabilities is challenging as the underlying reasons of buffer overflow vary widely. This paper presents a novel method for BOF test for ANSI C language, which uses program instrumentation and mutation test technology to test the BOF vulnerabilities, on the basis of analyzing the invariants for BOF vulnerabilities. The implementation shows that it can check the attack of BOF vulnerabilities adequately and accurately, in the circumstances of no large losses in performance.**

Keywords-buffer overflow, invariant, mutation test, program instrumentation

I. INTRODUCTION

Buffer overflow (BOF) is one of the worst vulnerabilities. Statistical data in NIST [1] shows that BOF has occupied about 10~25% in all the number of vulnerabilities. The BOF vulnerabilities imply specific flaws in a piece of software, allowing attackers to overflow data buffers that might lead to the corruption of neighboring variables. If neighboring variables are sensitive (e.g., return address or frame pointer of a function), then BOF attacks might lead to program crashes (or segmentation violations) while trying to write illegal memory locations. Therefore, testing software implementations for BOF vulnerabilities is important. However, it is challenging as the causes of BOF vulnerabilities include the limitations of implementation languages and their associated libraries (e.g., ANSI C and its standard libraries), logical errors etc.

Traditional complementary approaches to test BOF vulnerabilities include source code auditing, static analysis, and runtime monitoring. Source code auditing is an expensive and time consuming process. Static analysis (e.g., ITS4) suffers from numerous false positive warnings. The runtime monitoring approaches augment executable programs to prevent exploitations. However, BOF vulnerability test helps fixing implementations early and decreasing losses incurred by the end users.

On the basis of analyzing the invariants for BOF, this paper presents a novel method for BOF test for ANSI C language, which uses program instrumentation and mutation test technology to do the BOF vulnerability test for software, hoping that this method could expose the BOF vulnerabilities rapidly, adequately and accurately.

II. MECHANISM

A. Program instrumentation

Program instrumentation is proposed by Professor J.G. Huang [2, 3], and it is widely used in practice, such as testing branch coverage and statement coverage. Similarly, program instrumentation can also be used to do the program security test.

Existing program instrumentation technology includes source code instrumentation (SCI) and object code instrumentation (OCI). OCI need not re-compile the program, so it is efficient and can be used to analyze the third part program without source code. However, it is difficult for OCI to implement, because it involves language's specific implementation and the underlying mechanism of operating system, which are always confidential. Comparing with OCI, SCI is more simple to implement but less efficient, because its main idea is to do lexical analysis and syntax analysis for source code, then do code instrumentation and calculate test data for program.

In this paper, SCI technology is suitable. Probes could be inserted into the source code to monitor program invariants. Then the security test for program could be implemented according to the invariants, for the changes of invariants mean that some abnormalities about running environment occur. For example, a Save-probe could be plugged in to save the invariant information, and a Check-probe could be plugged in to check whether the invariant information changes.

The ideal outcome is to complete an automatic or semi automatic program instrumentor, which could insert probes into the program, based on the lexical and syntax analysis for C source code via YACC.

The implementation of instrumentor has not yet completed, and in the experiment part of this paper, the code

978-0-7695-3836-5/09 $26.00 © 2009 IEEE
DOI 10.1109/CIT.2009.90

instrumentation is completed manually.

B. Invariant

Program invariant [4] is a kind of logic assertion which can be used to show some true properties during the program running, which can help software developers understand the process data structure and algorithm effectively, and which has important values for all stages of software engineering, such as software design, coding, proof, testing, optimization and maintenance. Particularly, at a certain point the program does not satisfy related invariant (or the invariant even does not exit) during running, which means that the program has an error or input error, and the error provide a basis for further discovery and exclusion of software vulnerabilities.

Program invariant can be obtained by dynamical detector, such as Daikon [5] and DIDUCE [6], or by the method of manual analysis. In this paper, we extract the common characteristic from an invariant set of BOF, and the common characteristic is called abstract invariant.

Definition (Abstract invariant). An abstract invariant is the common characteristic of an invariant set.

Some abstract invariants of BOF vulnerabilities will be given below.

Abstract invariant A. Input length ≤ buffer size. Here "input length" is the size of the input string. It is equivalent to doing boundary detection for buffer, to ensure that the input string does not cross the boundary of the buffer in memory.

Abstract invariant B. RA0≡RA1. Here RA0 stands for the return address at the beginning of function call, and RA1 stands for the return address before function returns. It is mainly used to detect stack buffer overflow, evidently, RA0≠RA1 stands for buffer overflow.

Abstract invariant C. AR≡BR. Here BR (before return) is the counter before function RET, and BR++ will be done before function RET. Accordingly, AR(after return) is the counter after function RET, and AR++ will be done by function-caller when function successfully returns. This abstract invariant is to ensure the correctness of the trail of function-call.

Obviously, A is a necessary and sufficient condition for safe buffer, while B and C are only necessary but non-sufficient conditions.

C. Detection algorithm

Because of the abstract invariants, we can generate the detection algorithms facilely, which are the direct description of the logical expressions of abstract invariants, which can be translated into the corresponding data structures and procedures.

According to abstract invariant A, we can save the size (called M) of buffer and when user input data into the buffer, get the size (called N) of the input string. If N≤M, it returns True, otherwise, returns False (buffer overflow). The pseudo code is shown below.

```
……
M=SaveBufferSize
.....
N=GetInputSize;
if(M<N)
        BufferOverFlow;
```

According to abstract invariant B, we can save the return address at the entrance point (called ii), and before returning function checks the return address (called jj). If ii≠jj, it returns False (buffer overflow). The pseudo code is shown below.

```
……
ii=EntrancePointRET
.....
jj=GetCurrentRET;
if(ii≠jj)
        BufferOverFlow;
```

According to abstract invariant C, we can do BR++ before function returns, and do AR++ after function returns. And then we need another thread to compare BR with AR. If BR=AR, it returns TRUE, otherwise, if in the next two comparisons, BR>AR, returns FALSE. The pseudo code is shown below.

```
Thread 1:
    int f(……){
        ……
        BR++;
    }
    .....
    f(……);
    AR++;
Thread 2:
    if(BR>AR)
    if in the next two comparisons, BR>AR, returns FALSE.
```

D. Solution

A detection algorithm is executable, and the solution is logically equivalent to the corresponding abstract invariant; therefore, a solution can be generated from a abstract invariant.

Figure 1 illustrates the solution generating framework, in which the emphases are to determine the abstract invariants, to implement the corresponding algorithms and to test the program via mutation test technology.

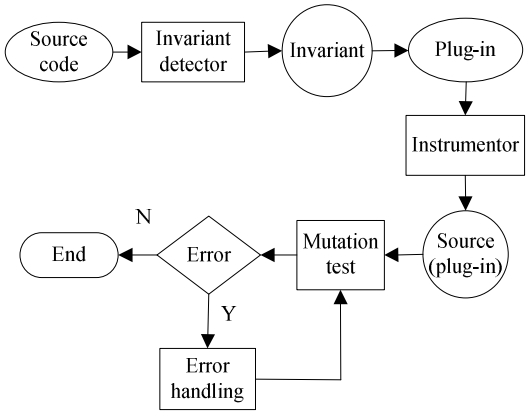

Figure 1. Framework of solution

From figure 1, solution steps can be easily deduced, which are shown as follow.

(1) Generate a set which is composed of a number of invariants which are identified from BOF vulnerabilities by invariant detector such as Daikon or by manual analysis;
(2) Obtain abstract invariants from the invariant set generated in step(1);
(3) Translate the abstract invariants into the specific pseudo-code algorithm;
(4) Implement the pseudo-code algorithm, forming executable plug-in code;
(5) Insert the plug-in code into the source program.
(6) Test program via mutation test technology.
(7) If any error occurs, handle the error and turn to step (6);
(8) Otherwise, stop the test.

III. IMPLEMENTATION

A. Principle of Buffer overflow

BOF attacks aim at disrupting the function of the execution of prerogative procedures, thus attacker can obtain the authority to do something special. Especially, if the procedures run in high privileges, the whole host would be controlled by attacker [7].

If the procedure does not check user's input string, buffer may be crossed or overflowed, thereby the procedure stack would be disrupted, and program's execution trail would be robbed [8].

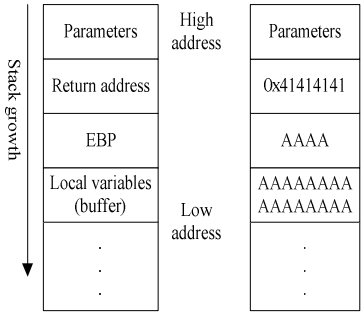

Figure 2. Overwrite the return address by overflowing buffer

At the time of function-call, processor pushes the function parameters, return address (the address of next instruction after function-call) and EBP into stack, and then uses current stack pointer ESP as the new base address. If there are local variables in the function, ESP will subtract a certain value to make place for dynamic local variables, including buffer variables. At the time of function return, processor pops EBP, and then pops return address to EIP to continue the running of the instructions after function-call statement.

In most cases the direction of stack growth is opposite to that of memory growth, but the same as the direction of procedures execution. So if the data inputted by user occupy more space than pre-allocated, stack will be overflowed, and three results--running failure, no impact or exploited by attacker to run his hazardous codes, may occur.

Figure 2 illustrates the most common example of BOF attack, in which a string of repeated 'A' is used to overflow the local buffer. In this case, the return address is rewritten as 0x41414141, which is to disrupt procedure's execution path, making it possible to run the hazardous codes. Generally, attacker puts a string which includes shellcode into buffer. The function's return address will be overwritten, making the new return address point to shellcode, which is shown in Figure 3.

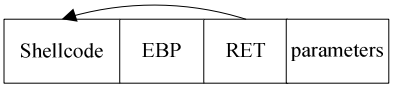

Figure 3. Constructing the dangerous string

B. Mutation test

Mutation test technology [9, 10, 11], which belongs to source class test, is an important method used to assess the effectiveness of test case. It can quantitatively analyze the adequacy of software security test cases, and it can also be used to test software vulnerabilities by controlling the test cases' adequacy manually, according to weighing the test environment and test requirements.

The key factor of using mutation test to do security test is to choose suitable mutation operators. For BOF vulnerabilities, the mutation operators can include mutating library function (e.g. replace strncpy with strcpy), mutating buffer size (e.g. increase or decrease the buffer size by one byte), mutating the format string and mutating the input string (e.g. remove the string's null character) etc.

C. Implementation of the test mechanism

We just do the implementation based on abstract invariant B, as the corresponding algorithm is similar to the other two deduced by abstract invariant A and C, and we can implementation the algorithms similarly.

In this work, probes are inserted into the function at the entrance and exit point based on abstract invariant.

Firstly, plug in codes are generated according to abstract invariant B in section 2.2. The pseudo-codes are shown as below.

```
void Fun (){
Save-module; //Save the return address
……
Check-module; //Check the return address
}
```

As Figure 2 shows, we can get return address by EBP, and the codes to get return address (Save-module) are shown in Table I.

TABLE I. TWO IMPLEMENTATION OF SAVE-MODULE

Linux(AT&T)	_asm_("
	movl %ebp, %eax
	movl 4(%eax), %edx
	movl %edx, $$
	");

Windows(Intel)	_asm{ mov eax, ebx

	mov edx, [eax+4]
	mov $$, edx
	};

The whole process of achieving BOF test is shown in Figure 4. At first, Save-probe saves the return address, and then Check-probe checks the return address and at the same time program does the mutation test. If no any error occurs, function will do nothing extra; otherwise function will report and handle the error and then do the mutation test again.

Secondly, because the abstract invariant B is a necessary but non-sufficient condition, we should ensure the test adequacy according to the mutation operators of BOF vulnerabilities, and in our work we choose the operators of mutating buffer size and mutating the input string.

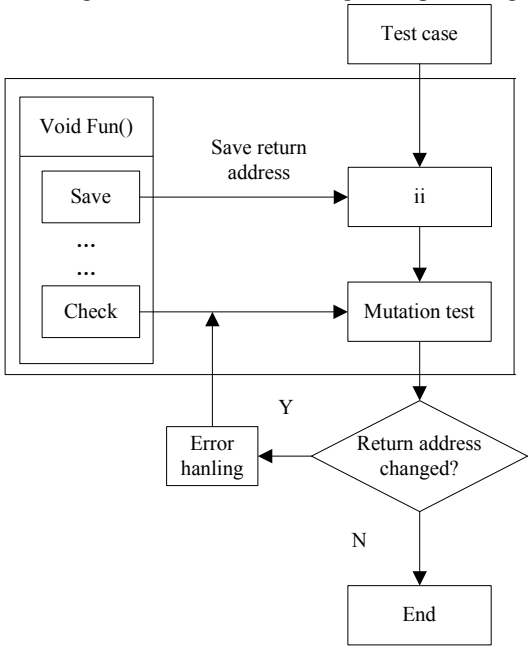

Figure 4. Framework for test

Conclusively, this method can ensure the adequacy of testing, and is able to test BOF vulnerabilities for program before published, which could reduce the loss, according to the use of mutation test method. At the same time, this method is transplantable—working normally in different systems.

IV. Experiment Results and Analysis

A. Correction analysis

A). Related work

Agrawal et al. [10] designed a comprehensive set of the mutation operators for ANSI C language which are applicable for program variables, constants, expressions and operators. However, there aren't any designed operators applicable for detecting BOF directly.

Hossain Shahriar et al. applied mutation testing for detecting buffer overflow vulnerabilities [11]. They designed 12 mutation operators including mutating ANSI C library function, the buffer size and the null character assignment. But it is too complicated to test BOF because it would take too much time to implement the 12 mutation operators. In particular, the test results are not apparent enough.

In this section, we just propose two mutation operators — mutating buffer size and mutating test case. And we apply the two operators to test Ftpdmin.

B). Ftpdmin

Ftpdmin is a minimal Windows FTP server without requiring an "install". It is intended to be run temporarily, on an as-needed basis to do file transfers between Windows computers without going thru the trouble of trying to configure windows file sharing or an FTP server, and also useful for Windows to Linux if the Linux computer doesn't happen to have an FTP server already set up and configured.

As a kind of open source software, we can get Ftpdmin from Internet, and the author allows us to do whatever we like with it.

Therefore, we decide to do the correction analysis by testing the buffer overflow vulnerability for ftpdmin by mutating technology, in MS Windows XP.

C). Mutating buffer size

Firstly, we mutate function ProcessCommands (Inst_t * Conn) --replace buf[MAX_PATH+10] with buf[100]. As shown in Figure 5, when we open a file or folder whose name consists of 106 characters (including NULL), a BOF error occurs, because the return address is overwritten by 'a' and NULL character.

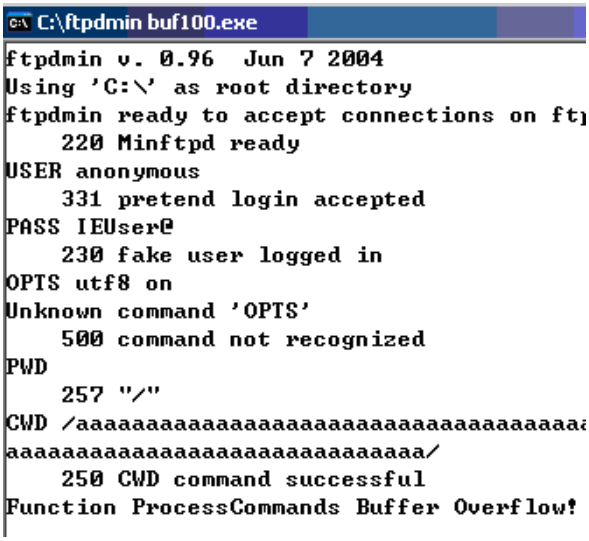

Figure 5. BOF test by mutating buffer size

Secondly, when we rename a file or folder to the name consisting of 106 characters (including NULL), the similar BOF error occurs because of the same reason above.

D). Mutating test case

Firstly, mutate the test case —replace 277 characters with 107(including NULL character).

Secondly, implement an exploit to send the mutated test case to Ftpdmin.

Finally, run the exploit, and then a BOF error is exposed,

as shown in Figure 6.

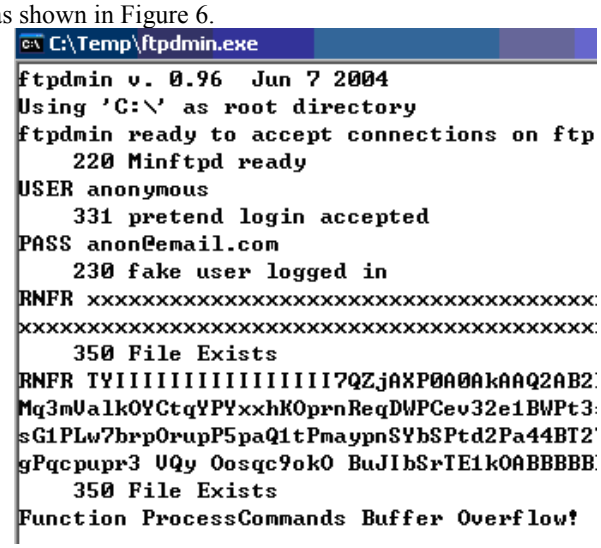

Figure 6. Buffer overflow test

In summary, the results in Figure 5 and Figure 6 show that we could test BOF correctly and apparently.

B. *Time analysis*

Because the additional time consumed by mutating can be neglected, we only test the following function test()'s time consumptions before and after code instrumentation, in the condition that there aren't any errors.

```
void test(){
   char buf[6];
   char a[]={"AAAAAAAAAAAAAA"};
   strcpy(buf,a);
}
```

In order to increase the precision of testing time, three for-loops are added to the program. Here, N is the number of i++ operation, controlling the inner loop in function test; T is the number of running function test (After a T loop, we compute the time once), controlling the middle loop; and M is the number of computing the time, controlling the outer loop.

The result shows that the time consumptions before instrumentation keep almost the same as those after instrumentation. After a number of tests, we record a group of overheads in the worst cases, as shown in Table II.

TABLE II. THE WORST ADDITIONAL OVERHEAD

(M,T,N) (10^x)	Before plug in(ms)	After plug in (ms)	Overhead (%)
(2,4,0)	0.000037894	0.000040321	6.402553
(2,4,1)	0.000067398	0.000070191	4.143756
(2,4,2)	0.000285771	0.000289552	1.323276
2,4,3)	0.002448070	0.002463941	0.648307
(2,4,4)	0.024689384	0.024771508	0.332625
(1,4,5)	0.251556604	0.253016158	0.580209

Table II shows that, the more time it takes to finish the function execution once, the less it affects the system due to the probes. For example, in our experiments when it costs 0.000037894ms to run function test once, the overhead is 6.40%, and when 0.251556604ms once, the overhead is 0.58%.

However, StackGuard makes performance decline about 10%~20%[6], and Visual Studio 2005 /GS makes performance decline about 5% in our system(from 0.000207425ms with /GS off to 0.000218881ms with /GS on). Therefore, we can say that our approach performs well.

V. CONCLUSION AND FUTURE WORK

This paper proposes a new BOF test method, which test BOF vulnerabilities via program instrumentation and mutation test technology, on the basis of analyzing the invariants of BOF. Now we have successfully achieved the test of BOF and the results show that our approach is not only simple, but also efficient.

Next, we would apply the method to more vulnerability categories, that is, we need find more suitable abstract invariants, which can be conversed to solutions, for more vulnerability categories, in order to improve the security test mechanism based on invariant. And finally, it is possible to implement a solution to test a lot kind of vulnerability categories.

REFERENCES

[1] NIST, National Vulnerability Database. http:/ / nvd.nist.gov, 2009.

[2] J.C.HUANG, An Approach to Testing, Department of Computer science, University of Houston, 1975

[3] J.C. HUANG, Program instrumentation and software testing. University of Houston, 1978

[4] Wang yong, zeng qingkai, Approach to protecting program running based on program invariant, Couputer Engineering and Design, 2008

[5] Michael D. Ernst, Dynamically Discovering Likely Program Invariants, PhD thesis, Dept. of Computer Science and Eng., Univ. of Washington, Seattle, Wash., 2000.

[6] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. ICSE2002, 2002.

[7] Cowan C, Wagle P, Pu C, Beattie S, Walpole J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. DISC EX, 2000

[8] Fayolle PA, Glaume V. A Buffer Overflow Study Attacks&Defenses, 2002

[9] J.H.Andrews, L.C.Briand, Y.Labiche, Is Mutation an Appropriate Tool for Testing Experiments, ICSE'05, 2005

[10] Micheal Ellims, Darrel Ince, Marian Petre, The Csaw C mutation tool: Initial Results, Third Workshop on Mutation Analysis (Mutation 2007), 2007.

[11] H. Shahriar and M. Zulkernine, Mutation-based Testing of Buffer Overflow Vulnerabilities, IWSSE, 2008