Identifying Buffer Overflow Vulnerabilities based on Binary Code

Shunli Ding, Jingbo Yuan
Northeastern university at Qinhuangdao
Qinhuangdao, China
dingsl@163.com, jingboyuan@hotmail.com

Abstract—**Buffer overflow attack is the most common and arguably the most dangerous attack method. The buffer overflow detecting will play a significant role in network security filed. Various solutions have been developed to address the buffer overflow vulnerability problem. The paper presents a method that combines static analysis with dynamic test. By using the method we can identify a lot of potential weakness locations. A buffer overflow vulnerabilities testing system was developed. Using the system some PE-format files and dynamic link library files are detected respectively. The experiment results show that the method is feasibility and availability.**

Keywords-network security; buffer overflow vulnerability; static analysis; dynamic test

I. INTRODUCTION

At present network security has become a serious problem. The buffer overflow vulnerability is one of the most dangerous and widely distributed software vulnerability. Therefore, it is important to find buffer overflow, especially those that are not discovered or not reported or reported but not receive much attention. Many techniques have been developed to prevent attacks due to buffer overflows [1-3].

There are two ways to find buffer overflow exploitation traditionally. One is to analysis source code of program, which is called static analysis, like literature [4]. And the other is to test a program while it is running, which is called dynamic test, like literature [5]. However, in fact, except for some open source code programs, most source codes are hardly to be obtained. Dynamic test does not require source code, but it is inefficient for no having any knowledge about the program's internal structure.

This paper presents a method that strikes a proper balance between static analysis and dynamic test to identify buffer overflow vulnerabilities in binary code without source code. The right combination of the two approaches can enhance the security level of an application by detecting more vulnerability and with higher precision.

II. THE METHOD FOR IDENTIFYING BUFFER OVERFLOW VULNERABILITIES

A. The detecting flow

The detecting flow is illustrated in the following Fig. 1.

First, the binary file is converted to assembly language file and then by analysis of assembly language syntax and processes, the overflow point information that possible exist vulnerability are extracted. Here the overflow point is the first address of the function or the code segment that occur overflow in binary file. The information includes the function call relations and the addresses of potential overflow points and the size of the source buffer and destination buffer.

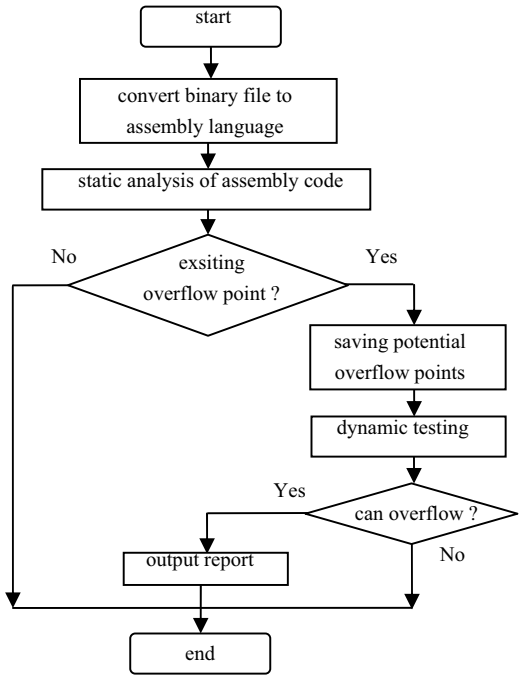

Figure 1. The detecting flow

In fact, software programs have standardized structure. A program is composed of code, stack and heap. The code is composed of many modules and each module generally realizes independent function. Modules store the local variable and can call each other. Although different programs may have different details, the structured programming model is suitable for various platform architectures. Therefore, this method can apply to all kinds of platform architecture and programming models. The paper takes the Intel 32 bits processor architecture.

B. Looking for potential overflow points

This stage analyzes the binary file, which generally refers to the PE form file and dynamic link library file on the WINDOWS platform. The method doesn't need to know any additional information about files or debugging information

978-1-4244-8728-8/11/$26.00 ©2011 IEEE

generated by compilation but only needs to know the entry address of the program. The purpose is to understand the basic structure of the test file and stack usage and function calling relations.

The binary file is first converted to assembly language and then analysis its instructions sequence to find the code segment that is responsible for copying the data in source buffer to the destination buffer, and then judges whether the code lacks buffer bound check. If lacks, it may have buffer overflow vulnerabilities. The main task of this step includes:

1) Determining the function call relations

All functions are identified by recursive analysis on object code. For executable programs the starting point of analysis is the entry of the program. For library files each exported function is as a starting point. The address behind each *CALL* instruction should be added to the function analysis table unless the address has been analyzed. In this way we can get a list of all the functions in the program.

2) Analysis stack space

The stack is used to store function arguments, local variables, or some information allowing to retrieval the stack state before a function call. This step is mainly to make certain whether there exist buffer in each function stack. There were two judging conditions. One is the size and layout of stack storage space, that is, the number of variables and the size of each variable. Another is the address type of pointer used when the stack seek. We can directly find the stack size on the functions.

Local stack space layout provides the number of local variables and the memory space information used by each variable. The information can be gained by analyzing the function code and collecting all quotation used the register *EBP* and negative offset locating. According to the instructions access on memory, the buffer size can be calculated and the buffer boundary can be determined.

For a general variable, if it is not used in a program the program will not reserved address space on the stack for it. But for a character array, as long as a member of the array is accessed the program will reserve address space on the stack for the array, and even if no instructions access to the rest address space in the compiled program. Moreover for a general variable, if it is just as a target operand and has never been used as source operand it will be optimized away by the compiler, so the program will not reserve address space on the stack for it. But for a buffer variable, even if it is not as source operand all along it will not be optimized away by the compiler. We can use these properties to determine the size of the buffer.

3) Analysis parameters

This step is to identify the parameter types of potential overflow functions. Pointer type parameter is particularly worthy of attention. If a function has not local buffer but receive a pointer as an argument, the pointer likely points to a buffer address of calling its function.

4) Use of local buffer

This step is used to detect functions with local buffer. By analysis the program code, the one hand to determine the sequence of instruction copy data to the local buffer by means of detecting whether has *loop* instruction that includes *MOV* instruction or *MOV* instruction with special prefix, on the other hand as long as detecting there is *PUSH REG*, *LEA REG* and *[EBP-OFFSET]* in front of *CALL* instruction, it can determine the buffer address as the instructions sequence of arguments calling.

5) Classifying functions

According to the results of above several steps we will classify functions into normal functions and potential overflow functions. The judgment rules are as follows:

- When the local buffer of a function is filled, if the buffer size is limited by constants the function will be marked as a potential overflow function.
- If a function takes over a pointer and the pointer is as the destination space address of the fill operation, the function will be marked as a potential overflow function.
- If a function takes over a pointer and it is passed as a parameter to another possible overflow function, the function will be marked as a potential overflow function.
- If a function with the local buffer transmits the buffer address as a parameter to another possible overflow function, the function will be marked as a potential overflow function.
- All functions that received pointers as parameters will be marked as a potential overflow functions.

It needs to pay attention to that it must give priority to functions that do not invoke other functions or do not provide pointers for calling functions.

C. Determining the overflow functions

The main work of the phase is to identify and test potential overflow functions in order to exclude unable overflow functions. As in the preceding stage we just roughly found some possible overflow functions, it needs still to determine what functions will or not overflow as being attacked. But it is very difficult to use a program to judge the correctness of another program, so it is necessary to use special test to exclude unable overflow functions.

The test procedure is similar to software robustness testing. We run target program with elaborate structured data to discover vulnerabilities within the program. But we only test functions marked as possible overflow in the previous step and not all function. This can save a lot of time. Moreover test data are based on previous found information in the program, instead of the parameter scope. This will also improve the efficiency of the test.

III. STATIC ANALYSIS

A. Static Analysis Tool

The paper chooses the IDA Pro as the disassembled tool. IDA Pro is a Windows or Linux hosted multi-processor disassembler and debugger[6]. IDA Pro is a professional disassembly tools and has the strongest ability to disassemble.

Because common buffer overflow vulnerabilities are caused by lack of verification on memory replication

operation etc, they have some regularity. We do disassembly process for the target code using IDA Pro and search the disassembling results to get potential system overflow risks, and then do analysis and discrimination to realize effective mining for system-level security vulnerabilities.

B. Script Controller

In addition to the extremely powerful disassembly functions and UI interactive interface, IDA Pro also provides IDC automatic scripting capabilities. Users can deal with disassembly database by writing automated scripts with specific purposes. IDC is an embedded Language. Its emergence greatly enhances the expansion of IDA so that many complex tasks may be completed by IDC.

C. Generating function call diagram

The function dependencies in executable binary file are firstly extracted by written IDC script. Then focus on analyzing on the calls that may cause buffer overflow and library functions of format string to determine if exist the security vulnerabilities in the program. Thus reduced vulnerability false alarms and improved the quality of vulnerability report. IDC script traverses functions in depth-first beginning from the program entrance, main function, of the target binary file. Simulate stack's FILO character and extract all functions in the program, where the called library functions in the program are not traversed to removal a lot of interference information and improve efficiency of analysis.

D. The implementation of static analysis

According to the above analysis the static analysis tool mainly completes the analysis on five potential overflow functions and records the function information to the database for subsequent dynamic testing. We rewrite five IDC script as following to identify and analyze the potential overflow functions:

Strcpy.idc: detect string copy function *Strcpy( )*;

Strcat.idc: detect string catination function *strcat( );*

Sprintf.idc: detect formatted string output function *sprintf( )*;

Repmov.idc: detect data copy of string operating instruction;

Movjmp.idc: detect data copy of *MOV +loop* instruction.

IV. DYNAMIC TEST

A. Detecting potential overflow function

Potential overflow function is a function without checking buffer border as accessing buffer. We use BugScam to detect the library function. The BugScam is able to detecting function as *MultiByteToWideChar, strcpy*, *strcat*, *sprintf*, *wsprintfA*, *lstrcatA*, *lstrcpyA* and so on. To facilitate the dynamic detecting, some information about function need to be recorded, so we modified IDC scripts. Aiming at the user-defined potential overflow functions there are mainly two cases.

1) Detecting string operating instruction

String operating instructions use *Rep MOVs* command to realize continuous write to the buffer, and its instruction and instructions sequence have obvious characteristic. The steps of detecting string operating are as following.

Step 1. Read an instruction. Is Rep movs instruction? If no, turn to step 1 or else continue.

Step 2. Is CX a constant? If is, turn to step 1 or else continue. Where register CX controls cycle times.

Step 3. Upward seek lea esi,xx or MOV esi,xx instruction, and calculate the size of source buffer

Step 4. Upward seek lea edi,xx or MOV edi,xx instruction, and calculate the size of destination buffer

Step 5. Save the initial address of program segment and size of the buffer

This process is repeated until every instruction test is complete.

2) Detecting MOV + LOOP instruction

The kind of potential overflow function achieves continuous write to the buffer through *MOV* instruction combined with *loop* command. In general to continuous write buffer, this type of sequence of instructions must meet the following several conditions:

C1: Consist of two or more than two *MOV* instructions.

C2: The source operand of one *MOV* instructions is the destination operand of another *MOV* instruction and belongs to register type.

C3: No other commands take the registers as target operands between the two *MOV* instructions.

C4: There are index registers in the *MOV* instructions and the index registers can be the same or different.

C5: Exist instruction adjusting index registers and the registers can be increment or decrement at the same time.

C6: These instructions are in the same loop.

Based on the above these conditions, the algorithm for recognition this type function is described as follows.

Step 1: Read an instruction beginning from the current address and analyze instruction type.

- If it is *MOV* instruction, records it in *MOV* instruction queue, and sets flag *FOUNDMOVS* when the number of *MOV* instructions is equal or greater than 2;
- If it is *INC, DEC, ADD* or *SUB* instruction, records in indexed address adjusting instruction queue, and sets flag *FOUNDINC* or *FOUNDDEC* depending on the situation;
- If it is *JMP* or *LOOP* instruction, records loop instruction and sets flag *FOUNDLOOP*.

Step 2: Do the following checks when the flags FOUNDMOVS, FOUNDINC (or FOUNDDEC) and FOUNDLOO all are set.

- Analysis if exist two *MOV* instructions to meet the conditions C2, C3 and C4. If no exist, turn to step 3.
- Analysis if exist instruction adjusting in indexed address adjusting instruction queue. If no exist, turn to step 3.

- Analysis the instructions coverage in *LOOP* instruction. If all these instructions are within the scope of their addresses, then continue analyze, or turn to step 3.
- Analysis if *MOV* instruction, index register adjustment instruction and loop instruction are in the same loop, if are in, turn to step 4, or turn to step 3.

Step 3: Clear FOUNDLOOP flag and read next instruction. If address is invalid then end or turn to step 1.

Step 4: Detecting succeed. Adjust the analyzed instruction address. If next address is invalid then end or turn to step 1.

B. The implementation of dynamic test

The implementation of dynamic test uses OllyDbg debugger[7]. OllyDbg is a user mode analyzing debugger. The paper does dynamic test using gained the potential overflow points information including address and buffer size. The testing process includes setting breakpoints and filling the destination buffer with different length data and judging truth of the overflow points according to the running results.

V. EXPERIMENT RESULTS AND ANALYSIS

A. Comparison test of the program with vulnerabilities

We developed a tool, called Testbufferoverflow, to perform the combined test of static and dynamic. The experiment firstly runs a program wrote with buffer overflow vulnerabilities and then uses Testbufferoverflow and Flawfinder[8] to respectively test the program and compare test results. The results are shown as table 1.

The tested program calls ten functions, which functions *strcpy, strcat* and *sprintf* are called respectively three times and in two of them the destination buffer is less than the source buffer. Another function uses *while* loop to copy data from the source buffer to the destination buffer. Therefore the correct results are function *strcpy, strcat* and *sprintf* have respectively two overflows and the *RepMOV* type has an overflow.

TABLE I. TESTING RESULT COMPARISON OF FLAWFINDER AND TESTBUFFEROVERFLOW

Function or Instruction Type	The correct results	The results of Flawfinder	The results of Testbufferoverflow
sprintf	2 overflows	found	Found and calculates the size of the source and destination buffer
strcpy	2 overflows	found	Found and calculates the size of the source and destination buffer
strcat	2 overflows	found	Found and calculates the size of the source and destination buffer
RepMOV	1 overflows	No found	found

When using Flawfinder to detect the program, it finds all library function and but don't find the overflow of *RepMOV* type. Using Testbufferoverflow, it finds six addresses of possible overflow in three functions and calculates the size of the source buffer and destination buffer. For third function call, because the destination buffer is large enough, so the function can't produce overflow and don't take them as overflow points. In addition, the tool also finds an overflow point of *RepMOV* type.

From the results we see that comparing with Flawfinder, our tool is more accurate and effective. It can find all the functions of the existing problems in the program and precisely locate all overflow points.

B. Testing for dynamic link library file

In the testing of dynamic link library file, each suspect function is directly called. The assigned values to the parameters come from the function information gained the previous step. The paper takes *Npdsplay.dll* as an example. Npdsplay.dll is a dynamic link library file of Media Player in Windows XP. Use the method and test tool of the paper proposed to detect buffer overflow vulnerabilities the file may exist.

1) Static test potential overflow points

After loading the script file *Run_analysis.idc*, the run results are shown as following table 2.

From table we can see that there may be buffer overflow at address of 100266ce. The other several addresses are likely to be false because it cannot figure out the size of the destination buffer.

2) Dynamic validity potential overflow

On the basis of previous testing results, set breakpoint at address 100266ce and find out the following code segment.

```
100266B3 8B75 10 mov  esi, dword ptr [ebp+10]
//esi point to the source buffer
100266B6 8B7D 14 mov edi, dword ptr [ebp+14]
//edi point to the destination buffer
100266B9 8946 08  mov dword ptr [esi+8], eax
100266BC 0FBE45 DA  movsx eax, byte ptr [ebp-26]
100266C0 8906   mov dword ptr [esi], eax
100266C2 0FBF45 D8 movsx   eax, word ptr [ebp-28]
100266C6 8946 04 mov  dword ptr [esi+4], eax
100266C9  8D45 DC   lea  eax, dword ptr [ebp-24]
100266CC 50   push   eax
100266CD 57  push   edi
100266CE E8 DDEBFFFF  call 100252B0
```

By analysis, we found that the register *eax* is pointing to the source buffer and the register *edi* is pointing to the destination buffer. We use a large amount of data to fill the destination buffer. If the size exceeds the size of the *eax* pointed area, overflow will occur and thereby cause run-time errors. Therefore, it can be determined that there is buffer overflow vulnerability.

TABLE II. TESTING RESULTS

Call address	Destination buffer	Source buffer	Describe
1001fe70	0	0	UNKNOWN_DESTINATION_SIZE: The analyzer was unable to determine the size of the destination; This location should be investigated manually.
10022371	-1	13	The maximum possible size of the target buffer (-1) is smaller than the minimum possible size of the source buffer (13). This is VERY likely to be a buffer overrun!
10022ce2	0	0	UNKNOWN_DESTINATION_SIZE: The analyzer was unable to determine the size of the destination; This location should be investigated manually.
10024ea2	0	0	UNKNOWN_DESTINATION_SIZE: The analyzer was unable to determine the size of the destination; This location should be investigated manually.
100266ce	22	24	The minimum size of the target buffer (22) is smaller than the maximum size of the source buffer (24). This is likely to be a buffer overflow problem!
100272a7	0	13	UNKNOWN_DESTINATION_SIZE: The analyzer was unable to determine the size of the destination; This location should be investigated manually.
100272c4	0	14	UNKNOWN_DESTINATION_SIZE: The analyzer was unable to determine the size of the destination; This location should be investigated manually.
10027cac	0	0	UNKNOWN_DESTINATION_SIZE: The analyzer was unable to determine the size of the destination; This location should be investigated manually.

C. *Testing for PE-format file*

The testing for an executable file is similar to the testing for a library file. The major difference lies in the testing process needs to intercept called external function. This is because the executable program has only one entrance so cannot derive functions like the library files.

In this part of the test we select 100 PE-format files in WINDOWS XP system to detect possible buffer overflow vulnerability, which include 50 .EXE files and 50 .DLL files. Test result shows in the following Fig.2

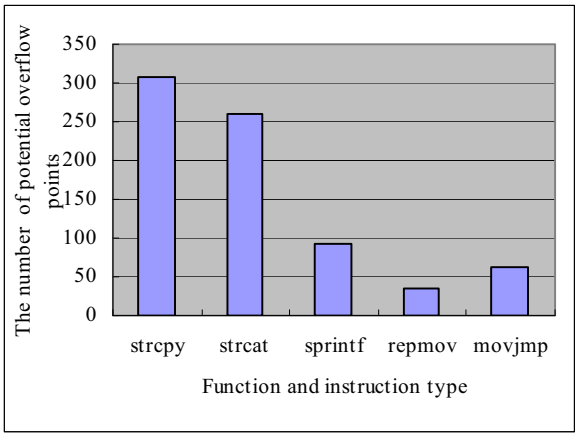

Figure 2. Test result of PE-format files

It can be seen that 756 potential overflow points are found out from 100 files. Average potential vulnerabilities for each file are less than 8, and *strcpy* and *strcat* types account for the majority of the vulnerabilities.

VI. CONCLUSION

Buffer overflow vulnerabilities are one of the most common vulnerabilities and are likely to continue to be a problem for a long time. The purpose of the paper is to study how to mine buffer overflow vulnerabilities that files may exist in the case of no knowing the source code. The paper takes a method combining static analysis and dynamic test. A system for detecting buffer overflow vulnerabilities based on binary code is developed. Using the system PE-format files and dynamic link library files are detected respectively. The system found and located potential overflow points, and excluded out the unusable overflow points by dynamic test. The test results illustrate that combining the two approaches-dynamic testing and static analysis, can complement each other and play a significant role in detecting the security vulnerabilities.

REFERENCES

[1] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," Proceedings of the 7th USENIX Security Conference, pp. 63-77, 1998.

[2] Fanping Zeng, Minghui Chen, Kaitao Yin, and Xufa Wang, "Research on Buffer Overflow Test Based on Invariant," Ninth IEEE International Conference on Computer and Information Technology, pp. 234-238, 2009.

[3] M. Akbari, S. Berenji, and R. Azmi, "Vulnerability detector using parse tree annotation," 2nd International Conference on Education Technology and Computer (ICETC), vol. 4, pp. 254-257, 2010.

[4] F. M. Puchkov and K. A. Shapchenko, "Static Analysis Method for Detecting Buffer Overflow Vulnerabilities," Programming and Computer Software, vol. 31, pp. 179-189, 2005.

[5] Seon-Ho Park, Young-Ju Han, Soon-Jwa Hong;et al. "The Dynamic Buffer Overflow Detection and Prevention Tool for Windows Executables Using Binary Rewriting," The 9th International Conference on Advanced Communication Technology, vol. 3, pp. 1776–1781,2007.

[6] http://www.hex-rays.com/idapro/

[7] http://www.ollydbg.de/

[8] http://sourceforge.net/projects/flawfinder/