Memory Vulnerability Diagnosis for Binary Program

Feng-Yi TANG, Chao FENG and Chao-Jing TANG

College of Electronic Science and Engineering, National University of Defense Technology, Changsha, China

Abstract. Vulnerability diagnosis is important for program security analysis. It is a further step to understand the vulnerability after it is detected, as well as a preparatory step for vulnerability repair or exploitation. This paper mainly analyses the inner theories of major memory vulnerabilities and the threats of them. And then suggests some methods to diagnose several types of memory vulnerabilities for the binary programs, which is a difficult task due to the lack of source code. The diagnosis methods target at buffer overflow, use after free (UAF) and format string vulnerabilities. We carried out some tests on the Linux platform to validate the effectiveness of the diagnosis methods. It is proved that the methods can judge the type of the vulnerability given a binary program.

1 Introduction

Memory vulnerabilities are difficult to detect and diagnose especially for those do not crash the program. Due to its importance, vulnerability diagnosis problem has been intensively studied. Researchers have proposed different techniques to diagnose memory vulnerabilities.

Jun Xu proposed a technique that uses the crash as a trigger to initiate an automatic diagnosis algorithm[1]. Moreover, the diagnosis process also generates a signature of the attack using data/address values embedded in the malicious input message, and is used to block future attacks. However, this technique needs an input that can trigger the program to crash and the validity of the generated signature cannot be verified.

Jiang Zheng combined both of the dynamic analysis techniques and static analysis techniques to solve automatic buffer overflow vulnerability diagnosis (BOVD) problem for commodity software[2]. However, the solution only targets at buffer overflow vulnerability and needs an effective exploit as input to finish the diagnosis process.

In this paper, we mainly focus on the memory vulnerabilities that might not lead to program corruption. Because of lack of source code, vulnerability analysis for binary programs have been a puzzle and the most popular solution may be fuzz test. But it is fair well known that the reported bugs are not always vulnerabilities. Therefore our work of vulnerability diagnosis is valuable for vulnerability fixing or exploitation and deserves attention.

2 Buffer Overflow Vulnerability Diagnosis

Buffer overflow vulnerabilities include heap overflow and stack overflow. Their theories are similar but the diagnosis methods to them are quite different.

2.1 Heap Overflow Diagnosis

A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as the `malloc()` call[3].

Heap overflows are usually just as dangerous as stack overflows. Besides important user data, heap overflows can be used to overwrite function pointers that may be living in memory, pointing it to the attacker's code.

It is comparatively easy to detect an overflow in the heap at runtime. When allocate an area in the heap, generally the memory allocator sets headers before or after

(or both) the memory area. These headers can contain some information like size, used or not, and etc.

Therefore, we can monitor the allocator so as to determine the base address and the size of the allocation areas. Then, check all STORE and LOAD accesses in memory to see which one is outside the allocated area.

In the diagnosis process, we supervise all calls to the `malloc` function and save the base address and the size of our buffer in a `std::list`. Then, when a STORE or LOAD occurs, we just need to check if the destination address is in our list. If it is outside our area, there probably is heap overflow.

2.2 Stack Overflow Diagnosis

A stack overflow condition is another buffer overflow condition, where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function) [4].

It is much more complicated to diagnose stack overflow than heap overflow though their theories are nearly the same. First, we should be aware that there are not one but several variables in the same stack frame. The main problem is to detect the overflow between these variables, which will probably not crash the program, that's why it's possible to miss this type of vulnerabilities without a binary analysis. So, it is quite difficult to detect the overflow between the variables at runtime without changing the code execution

There are four main steps to diagnose the stack overflow vulnerability: First, isolate the functions. Then, determine how many variables are instantiated in the stack frame. We use a `std::list` of stack frame which contains a unique ID and another `std::list` which contains all variables instantiated in this stack frame. Next, detect the loops which contain a STORE instruction to our stack frame. Generally a loop ends with a conditional jump. Finally, analyze this STORE to detect a potential overflow. If a STORE instruction occurs, we check where the value is stored. If the value destination is in our stack frame, we monitor that for each STORE the value is stored in the same area.

3 UAF Vulnerability Diagnosis

Use after free errors occur when a program continues to use a pointer after it has been freed [5]. Like double free errors and memory leaks, use after free errors have two common and sometimes overlapping causes:


```

0195584: mov dword ptr [rbp-0x8], 0x90909090 (rsp: 7fffc9e27ed0) (rbp: 7fffc9e27ed0)
ed0) (dest: 7fffc9e27ec8) (stack frame ID: 3)
0195591: mov dword ptr [rbp-0x8], 0x91919191 (rsp: 7fffc9e27ed0) (rbp: 7fffc9e27ed0)
ed0) (dest: 7fffc9e27ec4) (stack frame ID: 3)
[Write Mem B] 400507: mov word ptr [rbp-0xc], 0x91919191
counterWrite: 1
0195598: mov dword ptr [rbp-0x4], 0x0 (rsp: 7fffc9e27ed0) (rbp: 7fffc9e27ed0) (dest: 7fffc9e27ec0) (stack frame ID: 3)
[Write Mem B] 400523: mov byte ptr [rax], 0x45
counterWrite: 1
0195603: mov byte ptr [rax], 0x45
counterWrite: 2
[Write Mem B] 400523: mov byte ptr [rax], 0x45
counterWrite: 3
[Write Mem B] 400523: mov byte ptr [rax], 0x45
counterWrite: 4
[Write Mem B] 400523: mov byte ptr [rax], 0x45
400523: mov byte ptr [rax], 0x45 -- Stack overflow in 7fffc9e27ec8
counterWrite: 0
ld stack frame: 3
Num var: 3 ( 7fffc9e27ec4 0 7fffc9e27ec8 4 7fffc9e27ec0
ld )

```

Figure 4. Stack Overflow Diagnosis Result.

6 Conclusion

This paper mainly analyses the inner theories of major memory vulnerabilities and the threats of them. And then suggests some methods to diagnose the several types of memory vulnerabilities for the binary programs, which is a difficult task due to the lack of source code. The diagnosis methods target at buffer overflow, use after free (UAF) and format string vulnerabilities. We carried out some tests on the Linux platform to validate the effectiveness of the diagnosis methods. It is proved that the methods can judge the type of the vulnerability given a binary program.

References

1. J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," 2005.
2. "BUFFER OVERFLOW VULNERABILITY by Jiang Zheng M. S., Northeastern University, Boston, USA. 2001 Submitted to the Graduate Faculty of the Department of Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy U," 2008.
3. https://www.owasp.org/index.php/Heap_overflow
4. https://www.owasp.org/index.php/Stack_overflow
5. https://www.owasp.org/index.php/Using_freed_memory
6. https://www.owasp.org/index.php/Format_string_attack
7. M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and Analyzing Pointer Misuses for Sophisticated Memory-corruption Exploit Diagnosis."
8. M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and Analyzing Pointer Misuses for Sophisticated Memory-corruption Exploit Diagnosis."
9. G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi, "SafeStack: Automatically Patching Stack-based Buffer Overflow Vulnerabilities," pp. 1–14, 2013.
10. D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards Automatic Generation of Vulnerability-Based Signatures."
11. <http://triton.quarkslab.com>