Format String Bug Detection Using Memcheck

Yejin Yoon, Se-Hun Oh, and Mun-Kyu Lee

Abstract— Format string bugs (FSBs) are caused by careless use of the printf function in the standard C library and can cause abnormal actions in a program such as memory content leak or shell code execution. We modify a shared object in Memcheck which is one of the Valgrind tools for dynamic program analysis, and propose a solution for detecting FSBs. Our solution effectively detects an FSB by verifying whether the first argument of printf is allocated in the read-only data section or not. In other words, we can detect an FSB where the format string is not a read-only string, but a variable.

Research Keywords— Format String Bug, Memcheck, Valgrind

1 INTRODUCTION

A format string bug (FSB) [1] is a vulnerability caused by the programmer's careless use of a format string, i.e., the first argument, in the printf function in the standard C library. Using an FSB, an attacker can see and modify values in memory, and even can execute a shellcode [1]. For this reason, FSB should be detected and fixed correctly.

In this paper, we use Memcheck [2, 3] for detecting an FSB. Memcheck is a dynamic binary analysis (DBA) tool built on Valgrind [4, 5, 6], a dynamic binary instrumentation (DBI) framework on Linux. A DBA tool can analyze dynamically a program while it is running, so we can analyze the program according to its actual execution flow [5]. Memcheck can find memory leaks and memory errors in a program. In particular, Memcheck can detect invalid memory access in a program. In this paper, by slightly modifying a shared object in Memcheck, we add the FSB detection functionality to Memcheck.

2 PRELIMINARIES : *PRINTF* AND FSB

An FSB is caused by a programmer's careless use of the format string argument in *printf*. To understand how a format string is dealt with when *printf* is executed, we have to analyze the call path of *printf*.

In general, the *printf* call path begins with ld-2.23.so and finally ends with a *write* system call. See Fig. 1. Memcheck redefines many standard C library functions including *strchrnul* which is on the above call path and redirects the call to this new function. The redefined functions are included in modified shared objects, e.g. vgpreload_memcheck-amd64-linux.so.

We would like to prevent an FSB by further modifying *strchrnul*. First, we examine the operation of *strchrnul* inside *printf* in detail. When *strchrnul* is called on the call path from *printf*, it finds all occurrences of "%" in the format string, which is the first

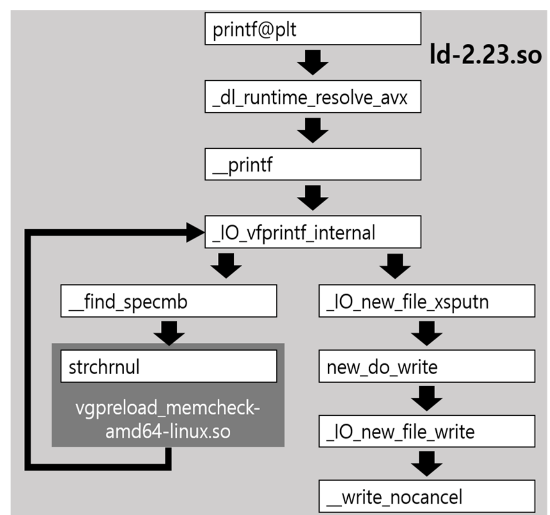

Fig. 1. Call Path of *printf* in Memcheck

- *Yejin Yoon is with the Department of Computer Engineering, Inha University, South Korea, E-mail: dnaygh78@inha.edu*
- *Se-Hun Oh is with the Department of Computer Engineering, Inha University, South Korea, E-mail: sehun.oh@cyberone.kr*
- *Mun-Kyu Lee (corresponding author) is with Department of Computer Engineering, Inha University, South Korea, E-mail: mklee@inha.ac.kr*

2014-2017 © ICRP. Published by the Institution of Creative Research Professionals

argument of *printf*. Then, the system copies the format string to a temporary buffer, replacing the occurrences of "%" with actual data specified by the second to the last arguments of *printf*. In a normal situation, the format string, i.e., the first argument of *printf*, should be placed in the read-only data (.rodata) section in memory because it should be hard-coded as a constant string by the programmer. If the first argument of *printf* is not a read-only string but it is either a local or global variable, an attacker may see and modify the memory values, or even execute a shellcode, which are called format string attacks. However, if we revise the inside of *printf* so that it does not allow a non-constant format string, we may prevent easily this kind of attack.

3 Proposed Method for FSB Detection

In this paper, we analyzed and modified the operation of Memcheck in Valgrind-3.11.0 version on Ubuntu 16.04 (64bits).

An FSB is caused by the programmer who use printf carelessly. Fig. 2 shows the normal usage of printf, which does not include any FSB. In this case, as we explained in section 2, the first argument of printf which is a format string is always allocated in .rodata section. In contrast, Fig. 3 shows an incorrect use of printf by the programmer. In this case, an FSB occurs and the first argument of printf is located in heap or stack, not .rodata section. We want to detect this situation. For detection of this kind of FSBs, we can use the memory layout of Memcheck. The memory layout of Memcheck is decided by the memory management policy of Valgrind. We found that Valgrind allocates heap and stack in the memory space with higher address values than that of library. In addition, the .rodata section is always located in the memory space with lower address values than that of library. Namely, the address of library is always greater than that of .rodata. If the first argument of printf tries to access the memory space for stack or heap, this program may be vulnerable to format string attacks. We can easily detect this situation by checking the address value of the first argument. That is, we can conclude that a program is potentially vulnerable to a format string attack if the address of the first argument is greater than that of the library. We decided to add this verification functionality to Memcheck.

In order to add the FSB detection functionality, we have to decide where to modify. As shown in Fig. 1, most of the functions are included in ld-2.23.so, but strchrnul is included in one of the shared objects of Memcheck, vgpreload_memcheck-amd64-linux.so.

```
#include <stdio.h>

int main(){
        char buf[100];
        gets(buf);
        printf("%s correctly printed\n", buf);
}
```

Fig. 2. Example code of using *printf*

```
#include <stdio.h>

int main(){
        char buf[100];
        gets(buf);
        printf(buf);        // incorrect use of printf function
}
```

Fig. 3. Example code of using *printf* incorrectly

```
shpik@shpik-400-470kr:~/val$ ./inst/bin/valgrind ~/malform/fsb
==16088== Memcheck, a memory error detector
==16088== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==16088== Using Valgrind-3.13.0.SVN and LibVEX; rerun with -h for copyright info
==16088== Command: /home/shpik/malform/fsb
==16088==
Hello
**16088**
**16088** *** format string bug detected ***:   at 0x4C303CC: VALGRIND_PRINTF_BACKTRACE (valgrind.h:6818)
==16088==    by 0x4C34895: strchrnul (vg_replace_strmem.c:1381)
==16088==    by 0x4E87207: __find_specmb (printf-parse.h:108)
==16088==    by 0x4E87207: vfprintf (vfprintf.c:1312)
==16088==    by 0x4E8F898: printf (printf.c:33)
==16088==    by 0x40058F: main (in /home/shpik/malform/fsb)
Hello==16088==
==16088== HEAP SUMMARY:
==16088==     in use at exit: 0 bytes in 0 blocks
==16088==   total heap usage: 2 allocs, 2 frees, 2,048 bytes allocated
==16088==
==16088== All heap blocks were freed -- no leaks are possible
==16088==
==16088== For counts of detected and suppressed errors, rerun with: -v
==16088== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
shpik@shpik-400-470kr:~/val$
```

Fig. 4. Result of detecting FSB with Memcheck

We inserted a code fragment in strchrnul to examine the memory address of the first argument of printf. Our code checks if printf accesses .rodata appropriately.

After adding a solution to Memcheck, we tested our modification using a sample code of FSB shown in Fig. 3. We executed the modified Memcheck on a computer with Ubuntu 16.04. According to our experimental result, the proposed solution using Memcheck detects an FSB successfully. As shown in Fig. 4, our modification prints the string "***format string bug detected***" when it detects an FSB.

4 Conclusions

In this paper, we proposed FSB detection using modified Memcheck. Our proposal is based on verification of the memory address value of the first argument in printf. However, this approach cannot detect an FSB

involving global variables, because these variables are also located in the lower address region. We leave this issue as our future fork.

ACKNOWLEDGMENT

This paper was presented at platcon-17 (2017 International Conference on Platform Technology and Service) with the same title. This work was supported in part by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-2012-0-00646) supervised by the IITP (Institute for Information & communications Technology Promotion) and by the IITP grant funded by the Korea government (MSIT) [No.R7122-16-0077, Development of the 10G Level RegEx Packet Pro-cessing SW for the Security Intelligence].

REFERENCES

[1] K. Lhee, S. J. Chapin, “Buffer Overflow and Format String Overflow Vulnerabilities,” Electrical Engineering and Computer Science. Paper 96. http://surface.syr.edu/eecs/96

[2] N. Nethercote, J. Seward, “How to Shadow Every Byte of Memory Used by a Program,” ACM VEE 2007, pp. 65-74, June 2007.

[3] J. Seward, N. Nethercote, “Using Valgrind to detect undefined value errors with bit-precision,” Proceedings of the USENIX'05 Annual Technical Conference, Anaheim, California, USA, p. 2-2, April 2005.

[4] N. Nethercote, J. Seward, “Valgrind: A Program Supervision Framework,” Electronic Notes in Theoretical Computer Science, 89, No. 2, pp. 44-66, 2003.

[5] N. Nethercote, “Dynamic Binary Analysis and Instrumentation,” PhD Dissertation, University of Cambridge, November 2004.

[6] N. Nethercote, J. Seward, “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,” ACM PLDI 2007, pp. 89-100, June 2007.

Yejin Yoon Received the B.S. degree in Computer Science and Engineering from Inha University in 2016. She is cur-rently a student in the M.S. course in the Department of Computer Engineering at Inha University. her research interests are in the areas of information security and trusted execution environment(TEE).

Se-Hun Oh Received the B.S. degree in Computer Science and Engineering from Inha University in 2016. He is currently a senior engineer in Cyberone. His research interests are in the areas of penetration testing and web hacking.

Mun-Kyu Lee Received his B.S. and M.S. degrees in Com-puter Engineering from Seoul National University in 1996 and 1998, respectively, and his Ph.D. degree in Electrical Engineering and Computer Science from Seoul National University in 2003. From 2003 to 2005, he was a senior engineer at Electronics and Telecommunications Research Institute, Korea. He is currently a professor in the Department of Computer Engineering at Inha University, Korea. His research interests include information security and theory of computation.

.