Memory Corruption-Basic Attacks and Counter Measures

Article · November 2016

CITATION
1

READS
3,331

3 authors, including:

Manajit Pal
KIIT University
2 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Prashant Kumar Dey
EURECOM
11 PUBLICATIONS **8** CITATIONS
SEE PROFILE

ISSN 2321 3361 © 2016 IJESC

Research Article Volume 6 Issue No. 11

Memory Corruption- Basic Attacks and Counter Measures

Manajit Pal[1], Vaibhav Dokania[2], Prashant Kumar Dey[3]
School of Electronics Engineering, KIIT University, India[1, 2]
Digital Security, EURECOM[3]

Abstract:
Memory corruption techniques are one of the oldest forms of vulnerabilities to be exploited by attackers. It had been under constant research and scrutiny for a very long time. This paper covers the basics of the types of memory corruptions and focuses on the detailed analysis of stack-based buffer overflow exploitation using local variables. Some real-life exploits and scenarios are presented and some of the popular mitigation techniques are discussed.

Keywords: memory corruption, buffer overflow, attacks, vulnerability.

I. INTRODUCTION

Memory corruption bugs are one of the oldest problems in the history of computer security starting with the 1988 Morris worm, which exploited a stack based buffer overflows, weak password brute-forcing and "zero-day" exploits. AlephOne's *Smashing the Stack for Fun and Profit, Phrack Magazine 49(14), 1996*, publicised the technique and prompted a widespread development of attacks to exploit the vulnerability[1]. Attacks have been known since 30 years and have been heavily exploited for the last 15 years. Applications, written in languages like C/C++ where the lack of memory or type safety makes enables attackers to alter the program behaviour and potentially take complete control of the control-flow thus enabling them to exploit the memory. The solution to it would be to write the applications in a type safe language and use annotations to make the code more secure. Unfortunately, this is unrealistic in many cases as low-level features are required for performance-critical programs (e.g. - operating systems). To curb the exploitation, operating system vendors and compiler designers took many measures. Stack cookies, exception handler validation, Data Execution Prevention and Address Space Layout Randomization make the exploitation of memory corruption bugs much harder. These anti-exploitation patches now ship by default on most operating systems with some packages offering them as after-market add-ons. For the purpose of our work, we will be discussing different types of memory corruptions and how to exploit their vulnerabilities. The later sections will focus on the defence techniques and exemplify some real-life exploits and their impacts, many of which have not been completely mitigated at the time of writing this paper.

II. BASIC THEORY

Memory corruption exploitation refers to a type of attack where the hijackers maliciously change the control and execution flow of the program by altering or corrupting the memory space of the application by different attack vectors. There are different vulnerabilities, some of which are briefly discussed below.

A. BUFFER OVERFLOW

A buffer overflow occurs when a program tries to store data beyond the boundaries of the buffer thereby overwriting the adjacent memory locations. Such kind of mistakes occurs while writing codes mainly due to unfamiliarity with the language or the lack of attention to details. Programming languages like in C/C++ does not have automatic memory management systems like dynamic bounds check and automatic resize of buffers. String manipulation functions like strcpy, gets do not properly check the string length can potentially be the candidate for buffer overflow. Other library functions like memcpy that can copy incorrect parameters and zero sized mallocs are also vulnerable. Consider the following program-

```
// ...
char src[]="ABCDEFGHI";
char tmp_buff[5];
int password_checked;
// ..
strcpy(tmp_buff,src);
// perform some action on backup string tmp_buff
// ...
```

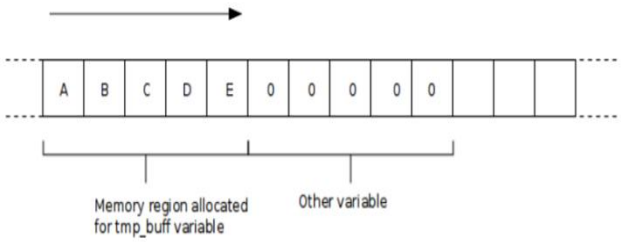

Figure.1. Vulnerable Program

Here we can that 5 memory allocation is made for the tmp_buff variable. The rest stores the garbage values (return address). When the function strcpy is executed, the memory or return address is overflowed by the variables in the src array. Once the return address has been modified the attacker has control of the program's control flow.

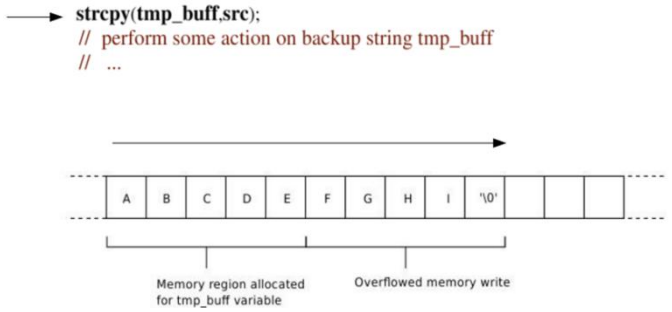

Figure. 2.overflow memory write

B. INTEGER OVERFLOW

Integer values have a range of 0 to 232 - 1 (for 32-bits unsigned integer) and -231 to 231 - 1 (for 32-bits signed integers). Now when we add more bits to the above range it would cause an overflow. For e.g.:

232 - 1 = 4294967295 + 1 = 0

As we can see the integer wraps around when the limit is exceeded [6]. Some languages like Java and ADA may throw exceptions during the process but many do not. Hence, the attackers can use this to supply large values to compute the size of a buffer that is malloc'ed and to access array (in particular the bound checks).

C. FORMAT STRINGS [6]

Apart from corrupting the memory an attacker can also read memory through their specified pointer leaks when that data is controlled by the output. One classic example is printf which can give memory access to the attacker if it is user controlled.

printf (user_controlled_data); // input %5$x prints the 5th // integer on the stack

Arguments can be pushed on stack and interpreted. For e.g.:

printf ("%s\n", err_msg); // attacker is able to leak arbitrary // memory contents by corrupting // err_msg pointer

Such format strings can be used to create invalid pointers which can read (or write) arbitrary memory locations.

D. STACK OVERFLOWS

A stack is a contiguous block of memory which is used by functions to store date for processing. Several operations are defined on stacks. Two important instructions are "PUSH", which puts data on the stack, and "POP", which removes data from the stack. It works on LIFO (Last In First Out) basis and expands towards lower memory addresses (on Intel, Motorola, SPARC, MIPS based systems). Every stack has a register called stack pointer (SP) which can point to the top or the adjacent free memory location depending upon its implementation. The stack consists of frames which are pushed which are pushed when a function is called and popped when the function is finished. In principle, a local variable can be referenced by giving their offsets to the SP. However, with more PUSHes and POPs, these offset change. Thus keeping track of these of the number of words requires multiple instructions or careful administration. Hence, for fast access to the variables frame pointers are used. The frame pointer is a register which points to a fixed location within a frame. The distance of the variables from the FP does not change with PUSHes and POPs.

Consider the following program [5]-

```
void vulstack(){
char buff[10];
scanf( "%s", buff);
}
```

Here the function vulstack defines a stack buff of 10 bytes which is located a few words before the return address and the frame pointer. Now the scanf() call might receive an arbitrarily large input which can overflow the buffer buff leading to the memory address of the frame pointer and the return address being tainted by the input data. This new address is used when the vulstack () returns. Thus we can see that the control of the program can be diverted by an attacker specified location which may contain instruction "shellcodes".

E. HEAP OVERFLOW

Heap overflow operates on a similar notion of stack overflow, i.e. an attacker can read/write beyond the bounds of a buffer. The major difference is that a heap does not hold a saved pointer to overwrite hence it gets harder to attack [4]. The overwriting of saved pointer in a heap or overwriting with exploitable values are fairly easy to understand but they cannot be defined as generic class attacks. Free memory chunks are managed in heap by a doubly linked list. Each allocated heap chunk includes meta-data for heap management [5].

III. EXPLOITATION

There are a vast number of attacks based on the above vulnerabilities, but for the purpose of our paper, we will focus on stack-based buffer overflow from/using local variables. The goal is to demonstrate how, in a simple program, overwriting a variable located in a process stack can give us access to a shell. Here we will use the technique to read contents of the .flag file. Let us first do a quick *whoami* and try to open the .flag file-

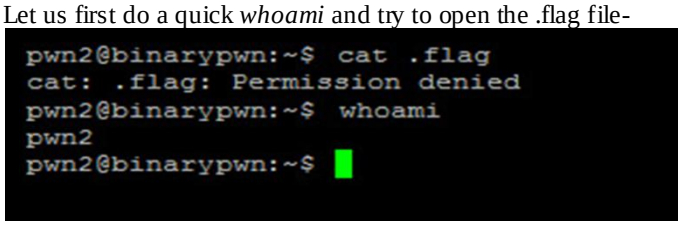

FIGURE. 3. TRYING TO ACCESS A NON-PERMITTED FILE

The code which we will use can be seen below in figure 4.

```
#include < stdio.h >
#include < stdlib.h >
#include < string.h >
void pwned() {
system("/bin/sh");
}
void main(int argc, char*argv[]) {
char customer_name[32];
int account_balance = 0;
if (argc != 2) return;
strcpy(customer_name, argv[1]);
printf("Hello %s ! Your account balance is %d\n ",
customer name, account_balance);
if (account_balance == 0xcafed0dd)
pwned();
printf("Bye");
}
```

FIGURE. 4. PROGRAM TO EXECUTE CODE FROM STACK

Running the above code through a normal sized input executes successfully without any error or warning which can be seen in the figure below-

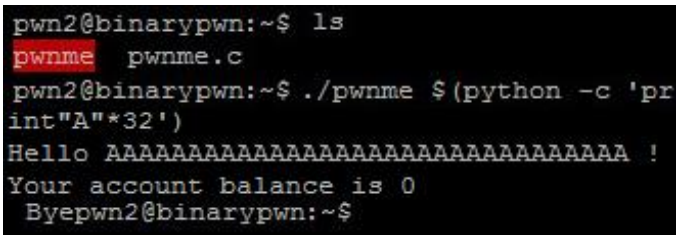

FIGURE. 5. RUNNING CODE WITHIN THE BOUNDS OF THE STACK

However, running the code with a large input creates segmentation fault and examination of the program in the debugger yields the following error-

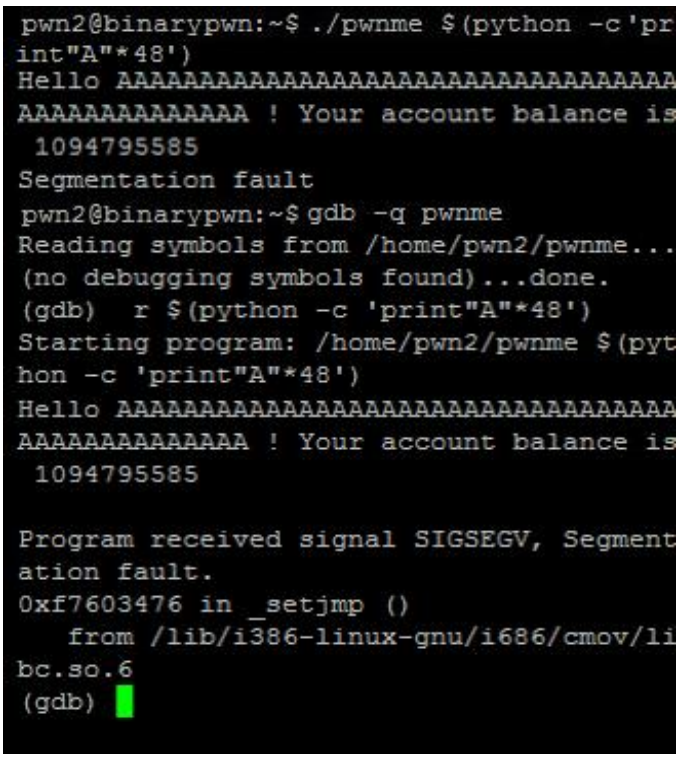

FIGURE. 6. RUNNING CODE BEYOND THE STACK BOUNDS AND DEBUGGING

One can clearly tell that there is a problem in *_setjmp()* library function on *0xf75f5476* which creates the error. Disassembling the program gives a better picture of the address of each instruction and how we can inject our desired address to get access to our shell.

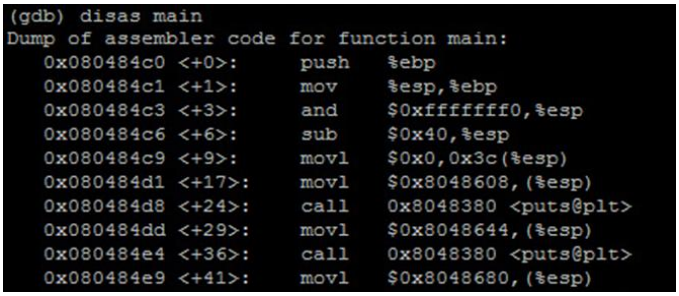

FIGURE.7.1.PROGRAM DISASSEMBLY 1

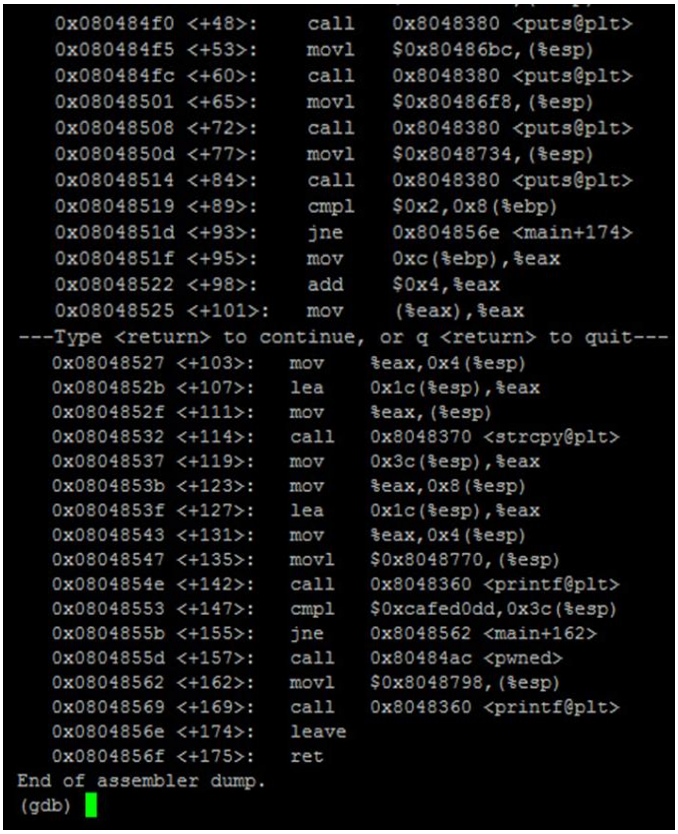

FIGURE.7.2.PROGRAM DISASSEMBLY 2

Thus on overflowing the input with the little endian format of our address in the input will create a stack overflow and in return will execute the *pwned()* function n return the result, thereby giving us access to the shell. Here, the little endian format of the address 0xcafed0dd is \xdd\xd0\xfe\xca. Let us now inject the same in place of "A" in the input of the program.

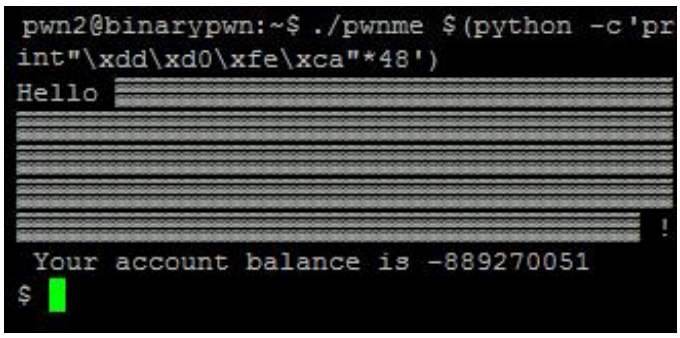

FIGURE.8.SHELL ACCESS

A quick *whoami* shows that we are now a privileged user. Hence, we have successfully exploited the stack overflow vulnerability. We can easily open the .flag to check its contents-

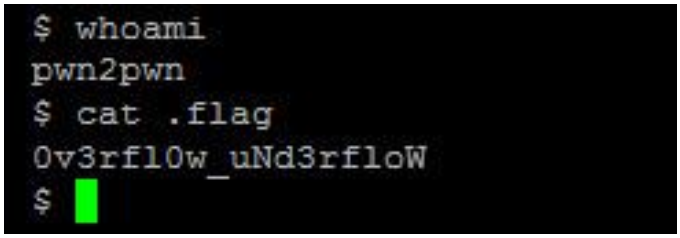

FIGURE.9.ACCESSING CONTENTS OF A NON-PERMITTED FILE

IV. REAL LIFE SCENARIOS

The win32 platform has been a home ground for most memory corruption attacks and malware, which forced Microsoft to deploy aggressive anti-exploitation methods in their recent Operating Systems. In 2004, Skylined's IFRAME Tag buffer overflow exploit used the technique of "Heap spraying" that creates a lot of heap blocks with NOP sleds followed by a shellcode [9]. It was mainly targeted to exploit internet explorer vulnerabilities. It was later found out that the same technique can be used to exploit software like Adobe Reader, Safari, Opera, Firefox and SQL Server.

Computer security researcher Alexander Sotirov released a paper titled *Heap Feng Sui in Javascript* in 2006, which refined the attack vector to give precise access to the heap for such attacks [9]. Many vulnerable software can also lead to information leaks. For e.g.: Easy chat server whose exploits are readily available from websites like exploit-db.com [8].

Apart from software, Operating Systems have also been prey to attacks related to memory corruption. Android, one of the most popular OS for smartphones has a serious bug in the heart of its code which is required to process videos. Stagefright, as it is popularly known due to the vulnerability in the *libStagefright* mechanism to automatically play the downloaded files [7]. However, patches have been rolled out since then to protect user data and to prevent memory corruption attacks resulting from software bugs.

V. MITIGATION

Several mitigation processes have been deployed which protect against unwarranted memory access and prevention of code redirection. Some of which are discussed below.

A. KERNEL ENFORCED PROTECTION

The kernel has the task of modifying the environment in which a program executes. It does so by changing the Layout of the process' virtual memory address space and by providing access control to avoid remote code injection into the memory [3]. Two methods are described below-

- **Memory Management Unit Access Control Lists (MMU ACLs) –** Non-executable memories are used for access control. A non-executable stack resides on a system where the kernel use "memory semantics" which includes separation of readable and writable pages, making all the stack and heap mapping anonymous and finally to deny conversions from executable to non-executable memory and vice versa.
- **Address Space Layout Randomization (ASLR) –** Exploits, in theory, rely on static values a known location of a buffer on a stack. These exploit techniques are defeated by inculcating randomness into the virtual memory layout for a particular process [2][6]. ASLR can produce randomness during the binary loading process so that all the library locations, binary mapping, stack memory, dynamic linking heap are all randomised cause the generic exploits to fail.

B. COMPILER ENFORCED PROTECTION

Compiler-enforced protection takes a completely different viewpoint while dealing with arbitrary code execution in a process. Special values may be inserted within the memories to make sure the buffer overflows are not able to reach the return address in the stack. Thus it ensures the integrity of the process control structure. Also rearranging the order of variables in the stack through pointers can be prevented by modifying the stack layout [3].

- **Stack canaries-** The primary goal is to prevent redirection of code execution to an attacker-controlled address. The addition of special canary values as a stack guard and checking of the canary value by modifications in the appendix function is an effective way to combat arbitrary code execution. Different types of canaries are used in Linux compiler-based protection as well as Microsoft Visual C++, .NET compiler protections. Some of these are Null canary, XOR canary, Random canary, Random XOR canary and Terminator canary.

Other mitigation techniques like control flow integrity, data flow integrity, code pointer integrity, etc. It may be noted that even though the techniques discussed are popular in their implementation, each of them have their limitations and can be exploited through advanced attacks like return-to-libc, copied code chunks, brute-forced/blind reverse oriented programming, JIT spraying, pointer inference and deference, heap spraying etc [6] and in worst case exploitation of software bugs to inject malicious codes.

VI. CONCLUSION

We have seen that most of the memory corruption errors are due low-level programming errors that allow attackers to corrupt memory and inject code thereby getting unauthorized access to systems and information leaks. There were times when big organizations claimed to have solved the problems of memory corruptions only to find new sophisticated attacks being spawned around them. The popular notion that a successful memory corruption attack is only possible by corrupting control data has been dissolved. A number of non-controlled data attacks can easily comprise the security of large network applications. There is a pressing need for research and development of publicly available software protection techniques. Taking advantage of the open source community, researchers can develop preventive measures that are compatible with different platforms and provide an impregnable solution to mitigate the problem.

VII. REFERENCES

[1] Aleph One, "Smashing the Stack for Fun and Profit." *Phrack Magazine*, 49(7), Nov. 1996.

[2] "PaX Address Space Layout Randomization (ASLR)."

[3] Silberman, Peter, and Richard Johnson. "A comparison of buffer overflow prevention implementations and weaknesses." *IDEFENSE*, August (2004)."

[4] Meer, Haroon. "AN EXAMINATION OF THE GENERIC MEMORY CORRUPTION EXPLOIT PREVENTION MECHANISMS ON APPLE'S LEOPARD OPERATING SYSTEM." *of publication: Proceedings of the ISSA 2009 Conference*. ISSA, 2009.

[5] Chen, Shuo, et al. "Defeating memory corruption attacks via pointer taintedness detection." 2005 *International*

Conference on Dependable Systems and Networks (DSN'05). IEEE, 2005.

[6] Szekeres, Laszlo, Mathias Payer, Tao Wei, and Dawn Song. "Sok: Eternal war in memory." *In Security and Privacy (SP), 2013 IEEE Symposium* on, pp. 48-62. IEEE, 2013.

[7] "How to Protect from StageFright Vulnerability". *zimperium.com*. July 30, 2015.

[8] EFS Easy Chat Server 3.1 - Stack Buffer Overflow (Online) https://www.exploit-db.com/exploits/33326/.

[9] Moshe Ben Abu, January 12, 2010, "Advanced Heap Spraying Techniques", *Recognize-Security*

VIII. ACKNOWLEDGEMENTS

We are thankful to pwnerrank.com for allowing us to use their server and the setup to test for various security measures under different scenarios. We are also thankful to KIIT University to allow using their resources for our research.