*Advances in Computer Science Research (ACSR), volume 73*
7th International Conference on Education, Management, Information and Computer Science (ICEMC 2017)

# Survey of Attacks and Defenses on Stack-based Buffer Overflow Vulnerability

Wei Wang
Information Engineering College, Capital Normal University, Beijing, China
wjwblabla@163.com

**Keywords:** Buffer overflow; Stack; Attack; Defense; C/C++

**Abstract.** With the rapid development of computer and related information technologies, risks associated with computer system are increasingly rampant[1]. And buffer overflow vulnerability is still the primary mean used by many hackers to attack soft application especially developed in unsafe programming languages like C or C++, although it has been exiting for two decades. A variety of corresponding defenses has been proposed and hackers also continue to come up with new attack methods to bypass the defense. This article mainly introduces the concept of stack-based buffer overflow and then discusses current main attacks and corresponding defenses based on the stack-based buffer overflow.

## Introduction

Although stack-based buffer overflow has been exiting for two decades, hackers still prefer to use it to exploit system at first step. Thus this article will focus on the principle of stack overflow vulnerability and discuss current main attacks and corresponding defenses.

Buffer holding data for a given type is a contiguous memory, which is distributed to program at runtime. A buffer overflow occurs when a program attempts to write content beyond the buffer's size to the buffer, resulting in any extra data to "overflow". This can enable an attacker to hijack control flow of the program thus making the program turn to execute other commands especially that attackers expect[2].

Stack-based buffer overflow is the most common form of buffer overflow and has been widely used to attack network and distributed system. In software, a stack-based buffer overflow occurs when a program write to a memory address on the program's call stack outside of the intended data structure which is usually a fixed-length buffer[3].

In order to understand why hackers prefer to use stack-based buffer overflow vulnerabilities at first to exploit system, let us look at the Fig 1.

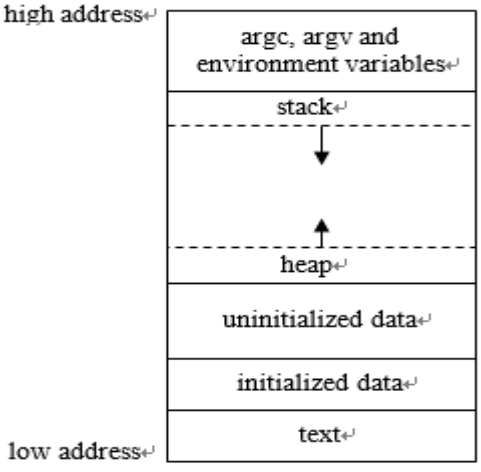


Figure 1. Exec memory layout

As Fig 1 shown, a stack is a contiguous and dynamic block of memory which is used by functions having a LIFO characteristic and growing from high memory address towards lower memory addresses on Intel based system. Stack is heavily used by functions. To hold function arguments and dynamically allocated space for local variables, each function call has its own independent buffer on stack called stack frame to maintain above information. And stack consists of stack frames which are created when a function is called and destroyed when a function is finished. In order to access the data on stack frame, system uses the EBP register pointing to the bottom of stack frame (high address) and the ESP register pointing to the top of the stack frame(low address).

As Fig 1 shown, the stack frame generally includes the following components associated with one subprogram call: argument variables passed on the stack, the return address, EBP, local variables and saved copies of any registers modified by the subprogram that need to be restored.

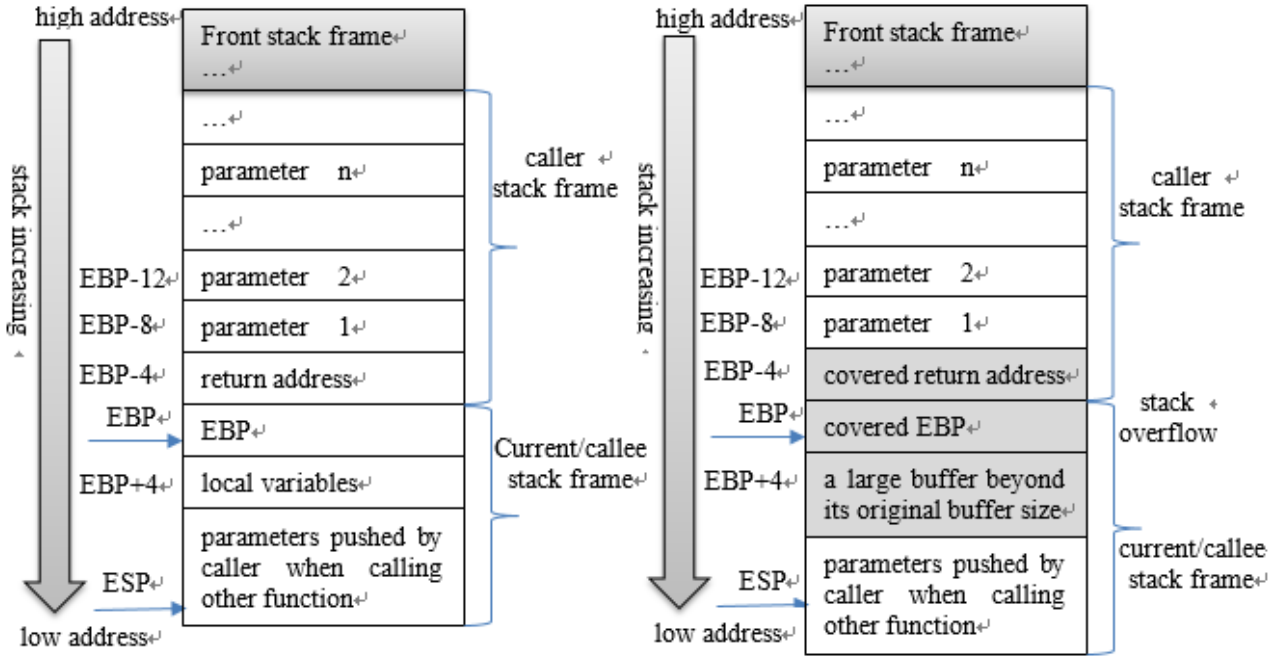


Figure 2. Stack frame layout VS. Overflowed stack frame layout

As shown in Fig 2, we can know that stack grows sequentially in contiguous memory, and both program data and control flow information are stored on the stack. It is thus possible for hackers to smash stack when a function copies data into a buffer on the stack frame without doing bounds checking, such as, strcpy(), memcpy(), gets(), etc. Thus, if the source data size is larger than the destination buffer size, this data will overflow the buffer towards higher memory address and probably overwrite previous data on stack, especially cover some controlling information such as return address(as shown in Fig 2) , thus result in the covered return address pointing to the attack code and then causing the attack code to execute. Thus stack-based buffer overflow vulnerabilities always allow an attacker to directly take control of the instruction pointer and, therefore, alter the execution of the program and execute arbitrary code, such as shellcode, for further attacks.

Next, we will discuss current main stack-based buffer overflow attacks in section2 and discuss corresponding common defenses in section3.

## Attacks on Stack-based Buffer Overflow

Stack-based buffer overflow attacks aim at disrupting the execution of somewhat privileged functions, which can allow an attacker to take control of the program, and then control the whole host. In general, attackers root program, and then perform somewhat "exec()" code to get the

execution shell with root privileges. For above purpose, a stack-based buffer overflow attack generally includes the flowing two steps: firstly arranging appropriate code which is usually the actual attack code in the program's address space, then secondly hijacking the control flow of the program according to a stack-base buffer overflow and make the program jump to the address where the attack code is.

**Arranging Attack Code in Program's Address Space.** There are following two methods to arrange attack code in the program's address space: injecting the attack code into program which is so called code-injection attack and reusing the already code as the attack code which is so called code-reuse attack.

Firstly, code injection is the exploitation of a computer bug that is caused by processing invalid data[4]. And this will cause an attack called code injection attack. Code injection attack refers to an attack by inserting a string (code) into an attacked program or system to interfere with its normal operation, and thus posing security threats[5]. The injected string can be a sequence of instructions which can run on the hardware platform. And attackers can inject the code into buffers anywhere, such as, stack, heap and static data area. Attackers usually attack the target system by exploiting the vulnerable input validation process.

Secondly, attackers can use exiting code of program to attack the program and this will cause an attack called code-reuse attack[6]. Code-reuse attacks including return-to-libc attack and return-oriented programming attack are software exploits in which an attacker directs control flow through existing code with a malicious result.

Return-to-libc stack is a computer security attack usually starting with a buffer overflow in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the process' executable memory, bypassing the NX bit feature(if present) and ridding the attacker of the need to inject their own code[7].

ROP(Return-oriented programming) is an effective code-reuse attack in which short code sequences ending in a ret instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called "gadgets"[8].

**Hijacking Control Flow of Program.** Attackers always hijack control flow of attacked program according stack-base buffer overflow by overwriting some control information on the stack frame and then divert the control flow to the attack code which is injected or reused. For above purpose, there are following five kinds of control information to overwrite in stack frame: return address clobbering, overwriting function pointer, overwriting exception-handler pointers (C++), overwriting long-jump buffers, overwriting saved frame pointer.

### Defenses on Stack-based Buffer Overflow

According to the analysis in both section1 and section2, in general, there are mainly three reasons which cause the stack-based buffer overflow attacks. First reason is that programmer may use some unsafe library functions, such as strcpy(),gets(),strcat() in C/C++ program language, which do not check the bound of buffer. Second reason is that there are not enough static and run-time boundary-checking mechanisms. And last reason is that the stack is executable.

For the first reason, we can use safe library functions instead of unsafe library functions to prevent stack from being overflowed.

In C/C++ language, now Microsoft offers some safe string handling library functions instead of unsafe functions which do not concern weather the buffer is oversized. For example, use strncpy(dst,src,dst_size-1) instead of strcpy(), fgets() instead of gets(), strncat() instead of strcat().

Table 1 Main attacks and corresponding defenses on stack-based buffer overflow

| Attack | code-injection | code-reuse |
|---|---|---|
| Defense | Canary, DEP and ASLR | ASLR |

Table 1 enumerates main attacks and corresponding defenses of stack-based buffer overflow. As Table 1 shown, for the second and last reasons which cause the stack-based buffer overflow attacks, there are three kinds of corresponding defenses: canary, DEP and ASLR.

**Canary.** Canary is a safe compiler technology to offer static and run-time boundary checking to prevent the stack from being attacked by code injection.

Canaries or canary words are known values that are placed between a buffer and control data on the stack to monitor buffer overflows[9].

Now, Microsoft compiler offers two methods to offer security checking at compiler-time and run-time. Firstly, Microsoft compiler offers /GS option, which inserts code to detect whether the return address is covered. Secondly, Microsoft compiler offers /RTCs option, which offers stack frame run-time error checking[10].

However, code-reuse attack can bypass this defense.

**DEP.** DEP(Data Execution Prevention) is a suite of software and hardware technologies, which can make the stack not be executable[11]. So, DEP can also prevent the stack from being attacked by code injection.

The principle of DEP is to set up a flag bit in the memory page of the system to mark the attributes of the memory page(executable).

Now, DEP mechanism also used in Microsoft compiler can make the data section and stack not be executable which can defense code-injection attacks.

However, code-reuse attack can also bypass this defense. So, to defend code-reuse attack, researchers propose defenses called ASLR.

**ASLR.** ASLR(Address Space Layout Randomization) is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities[12].ASLR randomizes the memory locations used by system files such as stack making it much harder for an attacker to correctly guess the location of a process which can defense the code-reuse attacks.

Address space layout randomization is a protection technology of buffer overflow, which can randomize the linear zone memory layout, such as, stack, heap and shared libraries to prevent attackers from direct positioning attack code location, so that ASLR can prevent stack from code-reuse attack, include both return-into-libc attack and ROP.

This mechanism is introduced since Windows Vista which can prevent stack from both code-injection attack and code–reuse attack.

The combination of DEP and ASLR creates a fairly formidable barrier for attackers to overcome in order to get attack code execution.

## Summary

With the rapid development of the Internet, it greatly facilitates people’s lives and national governance. As the network environment becomes more and more complex, attacks from the network are becoming more and more serious, and the attack based on buffer overflow is a common and serious attack. This paper expounds the principle of stack-based buffer overflow, and investigates the attacks and defenses based on stack-based buffer overflow.

Now, the research of attack and defense technologies based on buffer overflow is developing rapidly like an armament race.

## References

[1] Symantec. Internet Security Threat Report 2017, vol. 22. Technical report(2017)
[2] Buffer overflow on https://www.owasp.org/index.php/Buffer_Overflow
[3] Stack-based buffer overflow on https://www.owasp.org/index.php/Stack_buffer_overflow
[4] Code injection on https://en.m.wikipedia.org/wiki/Code_injection
[5] D. Ray, J. Ligatti: *Proc. POPL '12 Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*(Philadelphia,PA,USA, January 25-27,2012). 2012, p.179
[6] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen: *Proc. Symposium on Security and Privacy, SP 2013*(San Francisco,CA,USA, May 19-22,2013). 2013, p.574
[7] Return-to-libc on https://en.m.wikipedia.org/wiki/Return-to-libc_attack
[8] ROP on https://en.m.wikipedia.org/wiki/ROP
[9] Canaries on https://en.m.wikipedia.org/wiki/Buffer_overflow_protection#Canaries
[10] Microsoft compiler compiler-time and run-time security check on https://msdn.microsoft.com/zhcn/library/aa289171(v=vs.71).aspx#EIAA
[11] DEP on https://en.m.wikipedia.org/wiki/DEP
[12] ASLR on https://en.m.wikipedia.org/wiki/Address_space_layout_randomization