C++

Buffer Overflow the Security Threat. What Is It?

Contributed by Ahm Asaduzzaman

2003−08−01

advertisement

Article Index:

Buffer Overflow is an error to programmers as carrying the one is to elementary school math students. So what is the potential security risk, namely buffer overflow, that may arise due poor programming? Find out in this article by Ahm.

Modern computer architectures have an unfortunate design; it cannot make difference between data and instructions. If you can convince your program in someway, to run data that it has in memory, it will do it quite happily. A particular security flaw in computers, which has become almost ubiquitous in the last few years, is the buffer overflow. It is by far the most common security errors that programmers make.

It is common for few reasons: it is an easy error to make and hard to detect and by itself it has nothing to do with security, and another reason is, its human nature not to expect the unexpected. Buffer overflow attacks may be today's single most important security threat (approximately half of all security vulnerabilities) and most insidious data−dependant bugs known to mankind.

In this article an attempt has been taken to explain this security threat, an elementary knowledge of C, Assembly and debugging knowledge is required for a better understanding of topics in this article.

Definition of buffer overflow

Most application programs have fixed−size memory (buffers) that holds data. If an attacker sends too much data into one of these buffers and if program does not check the size of data, the buffer overflows. The computer may then execute the data that *overflowed* as if it were a set of command instructions. Now, if the exploitable buffer exists in a privileged process, a malicious program could then take full control of the server and can do any number of things of any types.

The Stack:

Before starting the technical details of *buffer overflow*, lets examine what happen when a program starts executing in memory. Before a program starts executing, operating system allocates memory space for it. The memory space allocated for a program is split into code segment (instructions that are to be executed by microprocessors), data segment (data is kept here) and stack segment (variables and temporary data are kept here). Lets write a simple C program.

```
#include <stdio.h>
int main()
{
   char Buffer[4];
   printf("%d",Add(5,6,7));
   return 0;
}
int Add(int a, int b, int c)
{
   a= a + b + c;
   return a;
}
```

Code Segment 1

A stack is a contiguous block of memory containing data and used whenever a function call is made. A stack pointer (**SP**) points to the top of the stack. In the example above, program will start executing from ***main ()*** block. When it calls ***Add*** function, following steps take place in the memory:

- Function parameters will be pushed onto memory from the right to the left (in this case 7, 6, 5 will be pushed onto stack).
- The return address pushed onto the stack followed by a frame pointer (address to be executed after the function returns, in this case after executing ***Add*** function). A frame pointer is used to reference the local variables (*Buffer [4])* and function parameters (*a, b, c*) because they are at a constant distance from the FP. Diagram 1 depicts different stack region during execution of this program. In most computer architecture stacks grow from higher memory address to lower and data are copied from lower to high address. For example, if **SP** pointed to memory address 0x0000FFFF, stack can hold 64−kilo bytes of data.

Low Memory **High Memory**

Buffer	**FP**	**Return Address**	**C**	**B**	**C**

−−−−−−> **Data Copies Up−−−>** <−−−−−−−−**Stack Grows Down <−−**

Diagram 1: During Execution of Program

The Threat

Skillful attackers might want to overwrite return address with the address of their own program, so it points back to buffer and execute the intended code literally hijack the processor's execution path. An example code that spawns a root shell in Unix systems can be found here. The root cause of buffer overflow problem is, that C/C++ is inherently unsafe. There are no bound checks on array and pointer references, that means, developer has to check the bounds, moreover, a number of unsafe string operations also exists in the standard C library, i.e., *strcpy(), strcat(), sprintf(), gets().*

Most Commonly, attackers exploit buffer overflow to get an interactive session on the machine. If the program being exploited runs with a high privilege level (such as root or administrator). Up to fifty percent of today's widely exploited vulnerabilities are buffer overflows. Read here, how Microsoft's programmers have been ordered to drop everything and work on fixing security problems.

Walkthrough Example I

Lets see the following example, if you execute this little program, you are prompted to introduce a string, if you type more than 4 characters, program will end with error message (like core dump). Because the string is bigger then the memory space allocated (char[4]) for it. This is the basic idea about buffer overflow.

```
#include <stdio.h>
 int main(){
 ReadMe();
}
int ReadMe(){
char Buffer[4];
scanf("%s", Buffer);
printf("%s",s);
}
```

Code Segment 2

Compile the above program.

```
>cc –ggdb T.C –o t

/home/asad$gdb ./t
GNU gdb 19991004

Copyright 1998 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or
distribute copies of it under certain conditions. Type, "show copying" to see the conditions. There is absolutely
no warranty for GDB.  Type "show warranty" for details. This GDB was configured as "i386−redhat−linux"...

(gdb) disassemble main

Dump of assembler code for function main:

0x80483f8 <main>:        push   %ebp
0x80483f9 <main+1>:      mov    %esp,%ebp
0x80483fb <main+3>:      call   0x8048404 <ReadMe>
0x8048400 <main+8>:      leave
0x8048401 <main+9>:      ret
0x8048402 <main+10>:     mov    %esi,%esi
End of assembler dump.
(gdb)

(gdb) disassemble ReadMe

Dump of assembler code for function ReadMe:

0x8048404 <ReadMe>:      push   %ebp
0x8048405 <ReadMe+1>:    mov    %esp,%ebp
0x8048407 <ReadMe+3>:    sub    $0x4,%esp
0x804840a <ReadMe+6>:    lea    0xfffffffc(%ebp),%eax
0x804840d <ReadMe+9>:    push   %eax
0x804840e <ReadMe+10>:   push   $0x8048480
0x8048413 <ReadMe+15>:   call   0x804830c <scanf>
0x8048418 <ReadMe+20>:   add    $0x8,%esp
0x804841b <ReadMe+23>:   lea    0xfffffffc(%ebp),%eax
0x804841e <ReadMe+26>:   push   %eax
0x804841f <ReadMe+27>:   push   $0x8048480
0x8048424 <ReadMe+32>:   call   0x804833c <printf>
0x8048429 <ReadMe+37>:   add    $0x8,%esp
0x804842c <ReadMe+40>:   leave
0x804842d <ReadMe+41>:   ret
0x804842e <ReadMe+42>:   nop
0x804842f <ReadMe+43>:   nop
End of assembler dump.
```

Now run the program and see what happens.

```
/home/asad$./t
AAA     − My input 3 characters
AAA     − Prints the output to the screen
```

Lets overflow the stack with 4 characters

```
/home/asad$./t
AAAAAAA  − My input 7 characters
Segmentation fault (core dumped)
```

The memory is overflowed; lets investigate this moment with GNU debugger gdb.

```
/home/asad$gdb t core
#0 0x8048400 in main ()
```

(gdb) info registers

```
eax     0x7    7
ecx     0x40071fd0    1074208720
edx     0x4010a980    1074833792
ebx     0x4010c1ec    1074840044
esp     0xbffffb58    −1073743016
ebp     0x414141      4276545
```

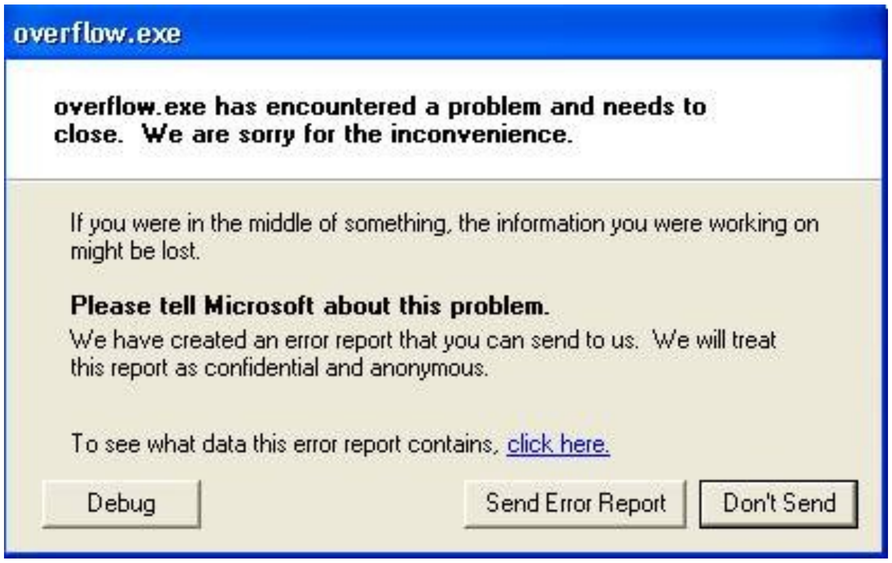
Here, you can see return address is now **0x414141** (which is a ASCII code for 3 consecutive characters of A) instead of **0x80483fb.**

NOTE: Windows programmers can download a free copy of Windows debugger (WinDbg) from Microsoft's site. In Windows system regardless of the type of memory dump that occurs, the dump file will be placed in %SystemRoot%\MiniDump. To see your path Open Control Panel −> Click System icon −> Select Advanced tab −> click on Startup and Recovery button.

Walkthrough

Example II

When you build and execute above program, the system will crash and you see the output as in Picture 1. The best way to understand it is visually (Diagram 2 and 3). Lets analyze the forensic evidence in the event of crash of this code snippet. When the *overflow* function is called from *main();* the code in *main()* will popup the string argument on the stack and call *overflow(), while* **EIP** indirectly place on the stack.



overflow.exe

overflow.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Picture 1

```
void overflow(char *str)
{
    char buffer[4];
   strcpy(buffer,str);
    return;
}
void handsup()
```

```
{
printf("your processor execution have been hijacked!\n");
    exit(0);
    return;
}
int main()
{
    char fat_buffer[]={'A', 'A', 'A', 'A',
                                    'B', 'A', 'S', 'E',
                                  '\x0','\x0','\x0','\x0'};
    void *funcPtr;
    unsigned long *lPtr;
    printf("Fat Buffer =%X\n", fat_buffer);
    funcPtr= handsup;
    lPtr=(unsigned long *) (
    overflow(fat_buffer);
     return 0;
}
```

Code Segment 3

Low Memory **High Memory**

Buffer	EBP	EIP (Ret. Add)		EAX	

−−−−−−> Data Copies Up−−−> **<−−−−−−−−Stack Grows Down <−−**

Diagram 2: Before strcpy ()

Low Memory **High Memory**

‘A’ ‘b’ ‘c’ ‘d’	‘e’ ‘f’ ‘g’ ‘h’	EIP 0x0000		EAX	

−−−−−−> Data Copies Up−−−> **<−−−−−−−−Stack Grows Down <−−**

Diagram 3: After strcpy ()

Imagine what would happen if the function to which we passed the oversized argument (*in our case fat_buffer []*) was in the kernel of operating system. You could jump to another function in the kernel. This would allow you to execute any code of your choosing and have it execute in kernel mode. Performing a successful buffer overflow attack can give you ownership of a machine.

In this article, we tried to understand the phenomenon of a security threat called Buffer Overflow, which causes mainly from bad programming practice. There are other type of stinky bug called *format string vulnerabilities* that has shocked the security community in the second half of 2000.

Format string vulnerability exist if a user can manipulate the format specification passed to a basic C function, such as printf, fprintf or Sprintf, vulnerability exists. These vulnerabilities represent a significant threat for servers and commercial applications and can be used to locally or remotely execute arbitrary code on a system.