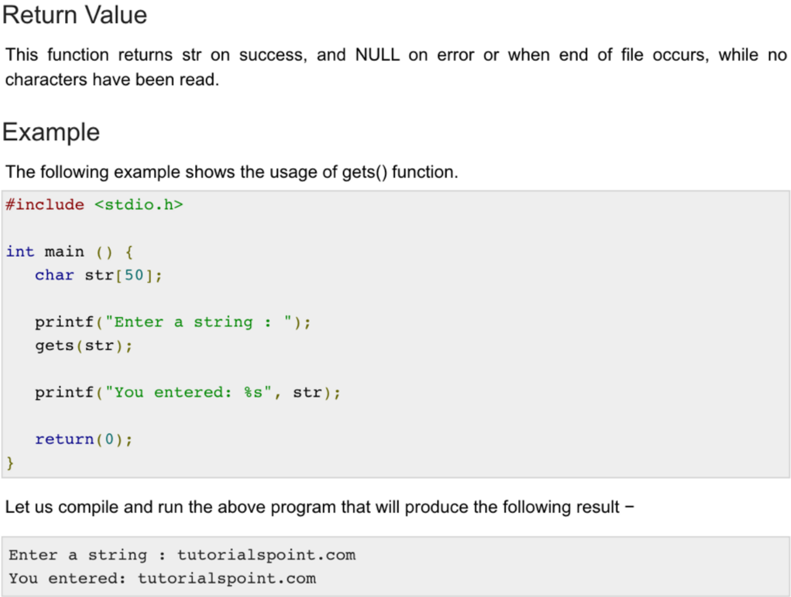


Return Value

This function returns str on success, and NULL on error or when end of file occurs, while no characters have been read.

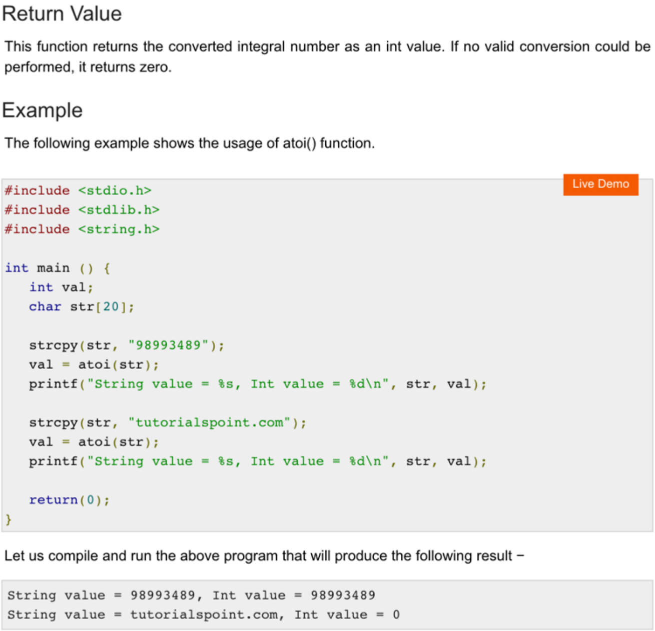
Example

The following example shows the usage of gets() function.

```
#include <stdio.h>

int main () {
   char str[50];

   printf("Enter a string : ");
   gets(str);

   printf("You entered: %s", str);

   return(0);
}
```

Let us compile and run the above program that will produce the following result −

```
Enter a string : tutorialspoint.com
You entered: tutorialspoint.com
```



Return Value

This function returns the converted integral number as an int value. If no valid conversion could be performed, it returns zero.

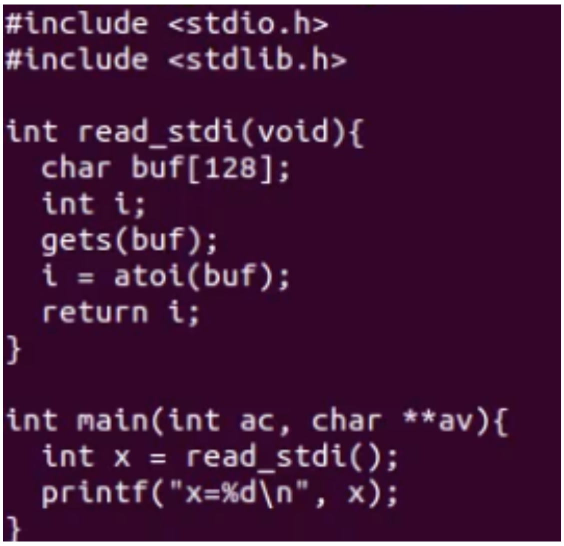
Example

The following example shows the usage of atoi() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main () {
   int val;
   char str[20];

   strcpy(str, "98993489");
   val = atoi(str);
   printf("String value = %s, Int value = %d\n", str, val);

   strcpy(str, "tutorialspoint.com");
   val = atoi(str);
   printf("String value = %s, Int value = %d\n", str, val);

   return(0);
}
```

Let us compile and run the above program that will produce the following result −

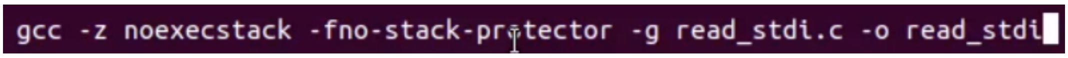
```
String value = 98993489, Int value = 98993489
String value = tutorialspoint.com, Int value = 0
```



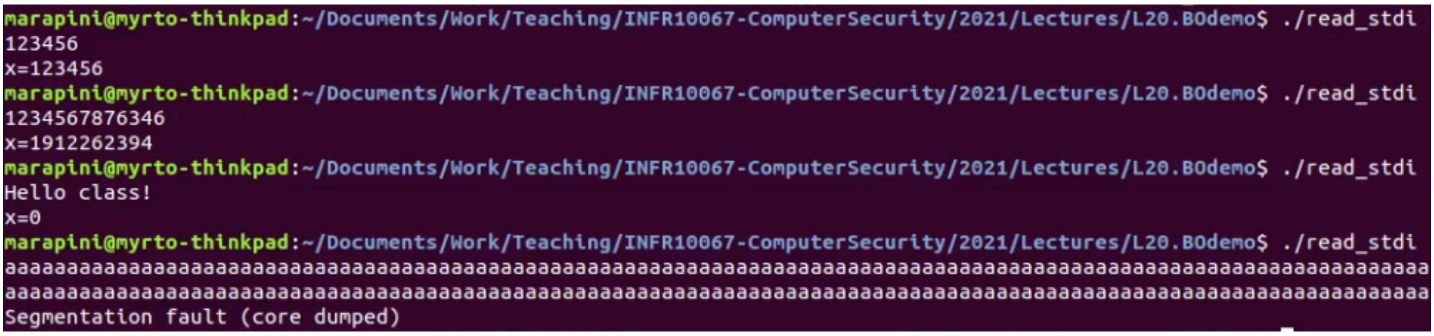
```
#include <stdio.h>
#include <stdlib.h>

int read_stdi(void){
   char buf[128];
   int i;
   gets(buf);
   i = atoi(buf);
   return i;
}

int main(int ac, char **av){
   int x = read_stdi();
   printf("x=%d\n", x);
}
```

We should allocate 128 bytes on the stack.

First, compile the code. Note that we have disabled the defense mechanism to allow out stack to be smashed. Compiling the code will give the following warning:
WARNING: the ‘gets’ function is dangerous and should not be used.



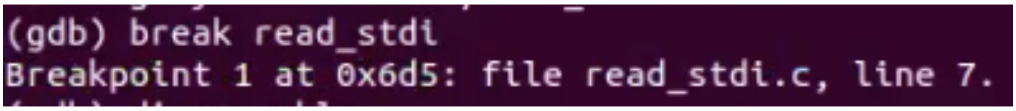
```
gcc -z noexecstack -fno-stack-protector -g read_stdi.c -o read_stdi
```



```
marapini@myrto-thinkpad:~/Documents/Work/Teaching/INFR10067-ComputerSecurity/2021/Lectures/L20.BOdemo$ ./read_stdi
123456
x=123456
marapini@myrto-thinkpad:~/Documents/Work/Teaching/INFR10067-ComputerSecurity/2021/Lectures/L20.BOdemo$ ./read_stdi
1234567876346
x=1912262394
marapini@myrto-thinkpad:~/Documents/Work/Teaching/INFR10067-ComputerSecurity/2021/Lectures/L20.BOdemo$ ./read_stdi
Hello class!
x=0
marapini@myrto-thinkpad:~/Documents/Work/Teaching/INFR10067-ComputerSecurity/2021/Lectures/L20.BOdemo$ ./read_stdi
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
```

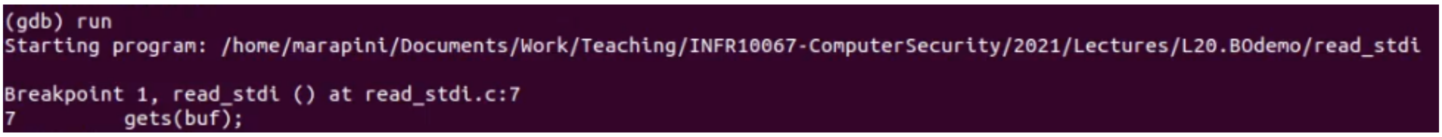
We use a debugger as we want to monitor the changes in memory areas.

First, set breakpoints.



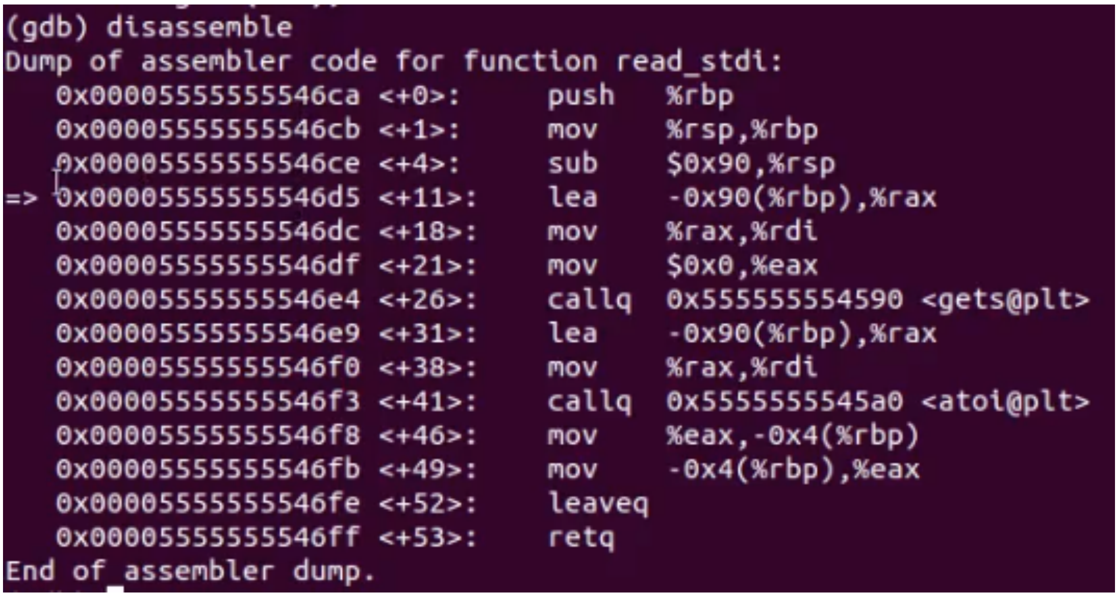
```
(gdb) break read_stdi
Breakpoint 1 at 0x6d5: file read_stdi.c, line 7.
```

Run the program and it will pause exactly at the first instruction in read_stdi.



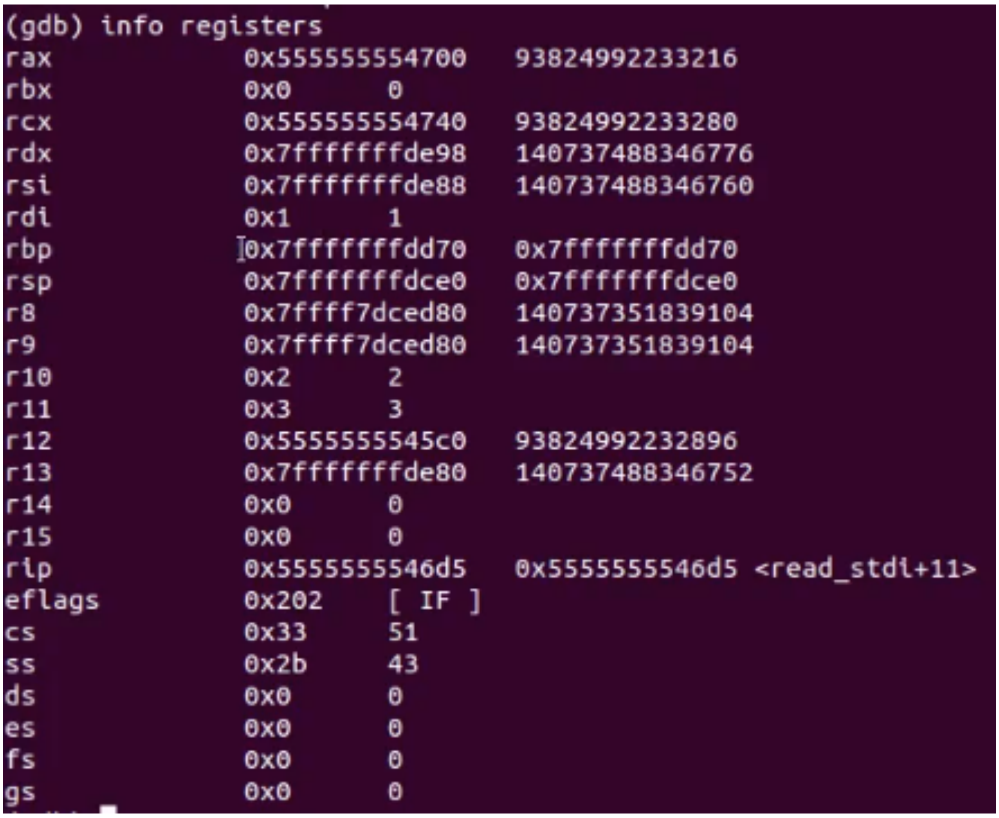
```
(gdb) run
Starting program: /home/marapini/Documents/Work/Teaching/INFR10067-ComputerSecurity/2021/Lectures/L20.BOdemo/read_stdi

Breakpoint 1, read_stdi () at read_stdi.c:7
7         gets(buf);
```

%rbp == %ebp (base pointer)
Subtracts 144 bytes from the stack pointer (making space for local variable i and the buffer).

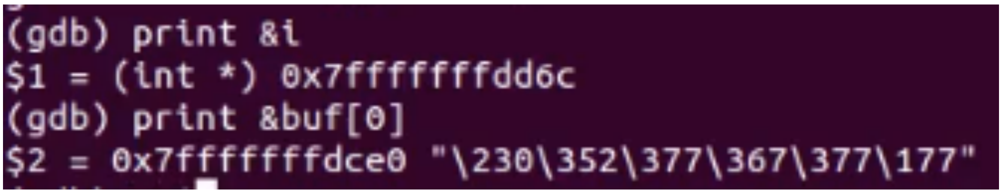


```
(gdb) disassemble
Dump of assembler code for function read_stdi:
   0x00005555555546ca <+0>:     push   %rbp
   0x00005555555546cb <+1>:     mov    %rsp,%rbp
   0x00005555555546ce <+4>:     sub    $0x90,%rsp
=> 0x00005555555546d5 <+11>:    lea    -0x90(%rbp),%rax
   0x00005555555546dc <+18>:    mov    %rax,%rdi
   0x00005555555546df <+21>:    mov    $0x0,%eax
   0x00005555555546e4 <+26>:    callq  0x555555554590 <gets@plt>
   0x00005555555546e9 <+31>:    lea    -0x90(%rbp),%rax
   0x00005555555546f0 <+38>:    mov    %rax,%rdi
   0x00005555555546f3 <+41>:    callq  0x5555555545a0 <atoi@plt>
   0x00005555555546f8 <+46>:    mov    %eax,-0x4(%rbp)
   0x00005555555546fb <+49>:    mov    -0x4(%rbp),%eax
   0x00005555555546fe <+52>:    leaveq
   0x00005555555546ff <+53>:    retq
End of assembler dump.
```

Stack pointer (rsp) is 144 bytes lower than the base pointer (rbp), as expected.

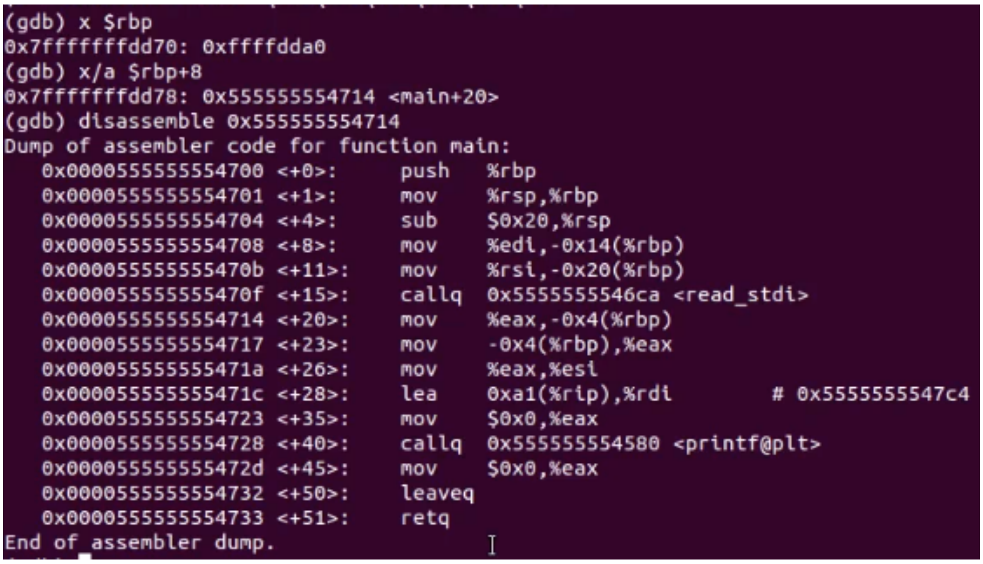


```
(gdb) info registers
rax            0x555555554700   93824992233216
rbx            0x0      0
rcx            0x555555554740   93824992233280
rdx            0x7fffffffde98   140737488346776
rsi            0x7fffffffde88   140737488346760
rdi            0x1      1
rbp            0x7fffffffdd70   0x7fffffffdd70
rsp            0x7fffffffdce0   0x7fffffffdce0
r8             0x7ffff7dced80   140737351839104
r9             0x7ffff7dced80   140737351839104
r10            0x2      2
r11            0x3      3
r12            0x5555555545c0   93824992232896
r13            0x7fffffffde80   140737488346752
r14            0x0      0
r15            0x0      0
rip            0x5555555546d5   0x5555555546d5 <read_stdi+11>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```



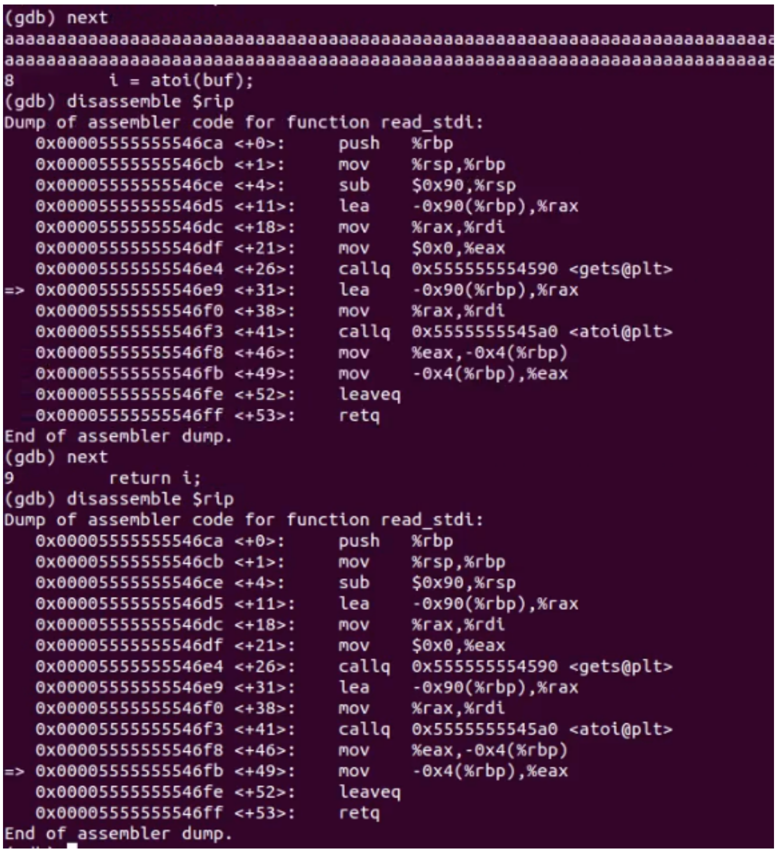
```
(gdb) print &i
$1 = (int *) 0x7fffffffdd6c
(gdb) print &buf[0]
$2 = 0x7fffffffdce0 "\230\352\377\367\377\177"
```

Look at what is stored in memory at the address where the base pointer points.
What is stored in dd70 is another address dda0 which is the base pointer of the previous stack frame. Return address lives 8 bytes above the base pointer, $rbp+8. As expected, it is at the point right below the call instruction.



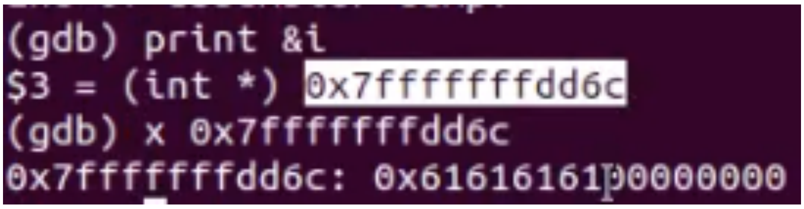
```
(gdb) x $rbp
0x7fffffffdd70: 0xffffdda0
(gdb) x/a $rbp+8
0x7fffffffdd78: 0x555555554714 <main+20>
(gdb) disassemble 0x555555554714
Dump of assembler code for function main:
   0x0000555555554700 <+0>:     push   %rbp
   0x0000555555554701 <+1>:     mov    %rsp,%rbp
   0x0000555555554704 <+4>:     sub    $0x20,%rsp
   0x0000555555554708 <+8>:     mov    %edi,-0x14(%rbp)
   0x000055555555470b <+11>:    mov    %rsi,-0x20(%rbp)
   0x000055555555470f <+15>:    callq  0x5555555546ca <read_stdi>
   0x0000555555554714 <+20>:    mov    %eax,-0x4(%rbp)
   0x0000555555554717 <+23>:    mov    -0x4(%rbp),%eax
   0x000055555555471a <+26>:    mov    %eax,%esi
   0x000055555555471c <+28>:    lea    0xa1(%rip),%rdi        # 0x5555555547c4
   0x0000555555554723 <+35>:    mov    $0x0,%eax
   0x0000555555554728 <+40>:    callq  0x555555554580 <printf@plt>
   0x000055555555472d <+45>:    mov    $0x0,%eax
   0x0000555555554732 <+50>:    leaveq
   0x0000555555554733 <+51>:    retq
End of assembler dump.
```

Type `next` to execute the next instruction.
`disassemble $rip` tells us where we are in the execution flow.



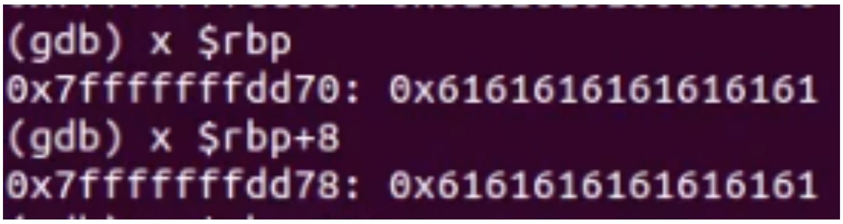
```
(gdb) next
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
8         i = atoi(buf);
(gdb) disassemble $rip
Dump of assembler code for function read_stdi:
   0x00005555555546ca <+0>:     push   %rbp
   0x00005555555546cb <+1>:     mov    %rsp,%rbp
   0x00005555555546ce <+4>:     sub    $0x90,%rsp
   0x00005555555546d5 <+11>:    lea    -0x90(%rbp),%rax
   0x00005555555546dc <+18>:    mov    %rax,%rdi
   0x00005555555546df <+21>:    mov    $0x0,%eax
   0x00005555555546e4 <+26>:    callq  0x555555554590 <gets@plt>
=> 0x00005555555546e9 <+31>:    lea    -0x90(%rbp),%rax
   0x00005555555546f0 <+38>:    mov    %rax,%rdi
   0x00005555555546f3 <+41>:    callq  0x5555555545a0 <atoi@plt>
   0x00005555555546f8 <+46>:    mov    %eax,-0x4(%rbp)
   0x00005555555546fb <+49>:    mov    -0x4(%rbp),%eax
   0x00005555555546fe <+52>:    leaveq
   0x00005555555546ff <+53>:    retq
End of assembler dump.
(gdb) next
9         return i;
(gdb) disassemble $rip
Dump of assembler code for function read_stdi:
   0x00005555555546ca <+0>:     push   %rbp
   0x00005555555546cb <+1>:     mov    %rsp,%rbp
   0x00005555555546ce <+4>:     sub    $0x90,%rsp
   0x00005555555546d5 <+11>:    lea    -0x90(%rbp),%rax
   0x00005555555546dc <+18>:    mov    %rax,%rdi
   0x00005555555546df <+21>:    mov    $0x0,%eax
   0x00005555555546e4 <+26>:    callq  0x555555554590 <gets@plt>
   0x00005555555546e9 <+31>:    lea    -0x90(%rbp),%rax
   0x00005555555546f0 <+38>:    mov    %rax,%rdi
   0x00005555555546f3 <+41>:    callq  0x5555555545a0 <atoi@plt>
   0x00005555555546f8 <+46>:    mov    %eax,-0x4(%rbp)
=> 0x00005555555546fb <+49>:    mov    -0x4(%rbp),%eax
   0x00005555555546fe <+52>:    leaveq
   0x00005555555546ff <+53>:    retq
End of assembler dump.
```

We give it a string, so the atoi function will return zero. We overwrote the integer i and the conversion function overwrote it again with zero (as expected).



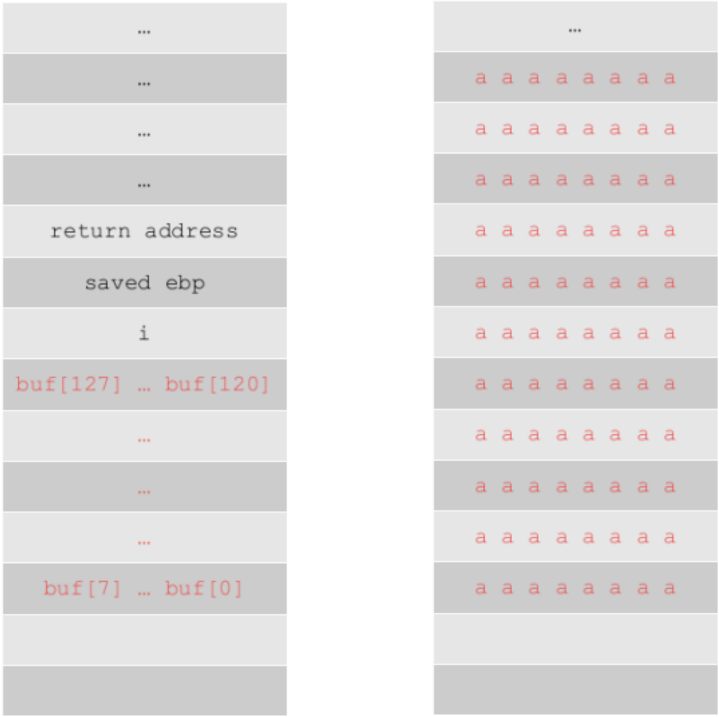
```
(gdb) print &i
$3 = (int *) 0x7fffffffdd6c
(gdb) x 0x7fffffffdd6c
0x7fffffffdd6c: 0x6161616100000000
```

We've overwritten the return address and base pointer. It now points somewhere random in memory; we're returning to an address which does not contain anything meaningful for the CPU

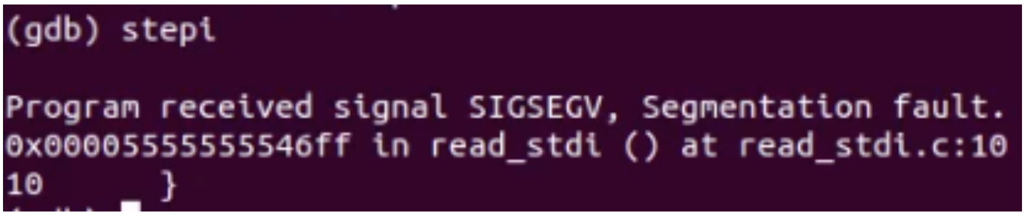


```
(gdb) x $rbp
0x7fffffffdd70: 0x6161616161616161
(gdb) x $rbp+8
0x7fffffffdd78: 0x6161616161616161
```

Therefore, the stack is smashed.
By providing some sort of input, you can overflow the buffer, which causes us to overwrite important parts of the stack frame and hijack the execution flow.



Eventually, the system crashes after executing the `retq` instruction.



```
(gdb) stepi

Program received signal SIGSEGV, Segmentation fault.
0x00005555555546ff in read_stdi () at read_stdi.c:10
10      }
```

How can we exploit buffer overflow in a useful way?

An attacker would want to somehow overwrite the return address not with garbage, but with the address of some code that he wants to execute.