Buffer Overflow Hacking Technique

By

Alan S H Lam

Buffer Overflow

Buffer Overflow is not a new hacking technique; however, it still haunts us today

Quoted from CERT:

> *Even though the cause [The Morris Worm of 1988] was highly publicized, buffer overflows are still a major cause of intrusions today. In fact, the first six CERT® Coordination Center advisories issued in 2002 describe buffer overflow flaws in several common computer programs*

Vulnerabilities due to buffer overflow

Name	Date
CAN-2004-0363: Stack-based buffer overflow in the SymSpamHelper ActiveX component	2004/03/19
CAN-2004-0362: Multiple stack-based buffer overflows in the ICQ parsing routines	2004/03/18
CAN-2004-0357: Stack-based buffer overflows in SL Mail Pro 2.0.9	2004/03/17
CAN-2004-0356:Stack-based buffer overflow in Supervisor Report Center in SL Mail	2004/03/17
CAN-2004-0353: Multiple buffer overflows in auth_ident() function in auth.c for GNU Anubis	2004/03/17
CAN-2004-0346 :Off-by-one buffer overflow in _xlate_ascii_write() in ProFTPD	2004/03/17
CAN-2004-0345: Buffer overflow in Red Faction client	2004/03/17
CAN-2004-0340: Stack-based buffer overflow in WFTPD Pro Server 3.21	2004/03/17
CAN-2004-0333: Buffer overflow in the UUDeview package for WinZip	2004/03/17
CAN-2004-0331: Heap-based buffer overflow in Dell OpenManage Web Server	2004/03/17
CAN-2004-0330: Buffer overflow in Serv-U ftp before 5.0.0.4	2004/03/17

Source: CVE http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow

Famous worms that make use of buffer overflow exploit

Name	Date
CA-2003-20 W32/Blaster Worm: buffer overflow vulnerability in Microsoft's Remote Procedure Call (RPC) implementation	2003/08/11
CA-2003-04 MS-SQL Server Worm	2003/01/25
CA-2002-27 Apache/mod_ssl Worm	2002/10/11
CA-2001-26 Nimda Worm	2001/09/18
CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL	2001/07/19

Source: CERT

Buffer Overflow

Buffer Overflow Exploit

In general, buffer overflow attack involves the following steps:

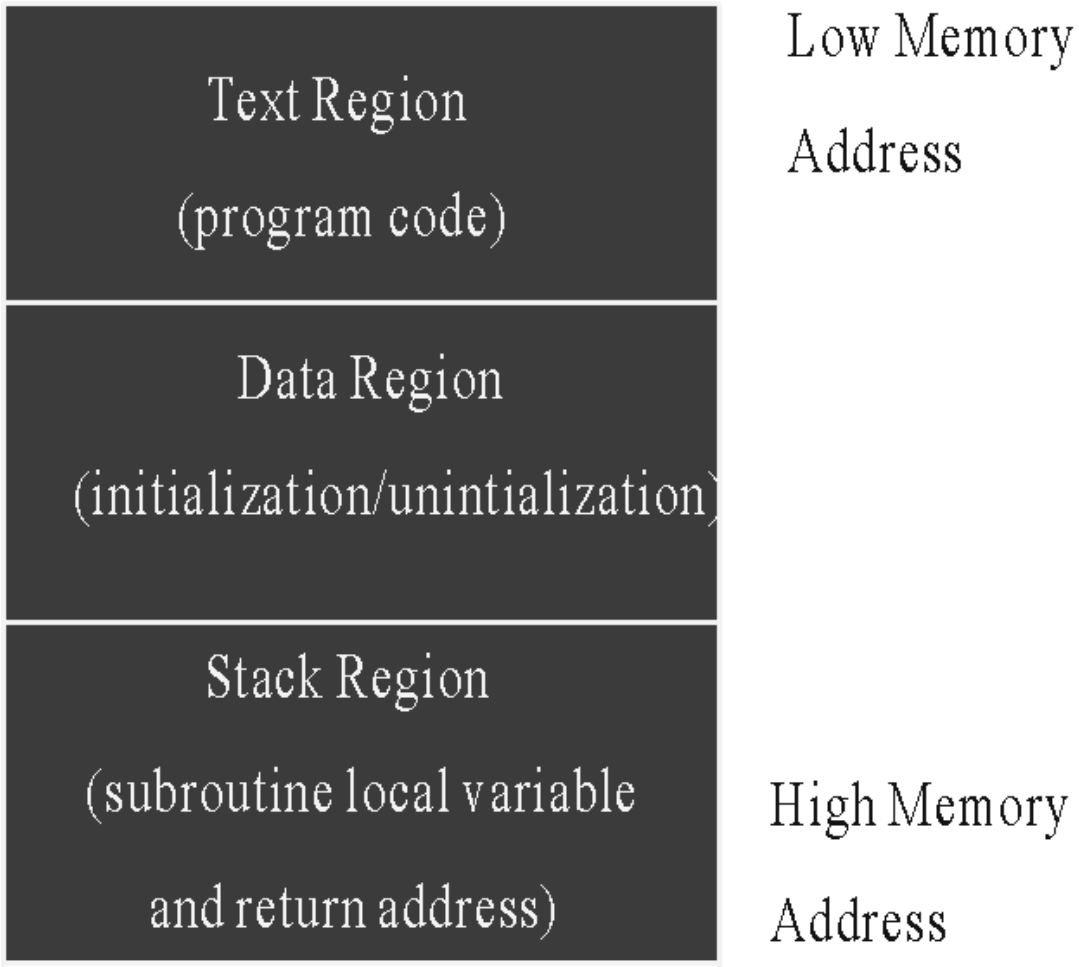
i. stuffing more data into a buffer than it can handle

ii. overwrites the return address of a function

iii. switches the execution flow to the hacker code

Buffer Overflow

Process Memory Region

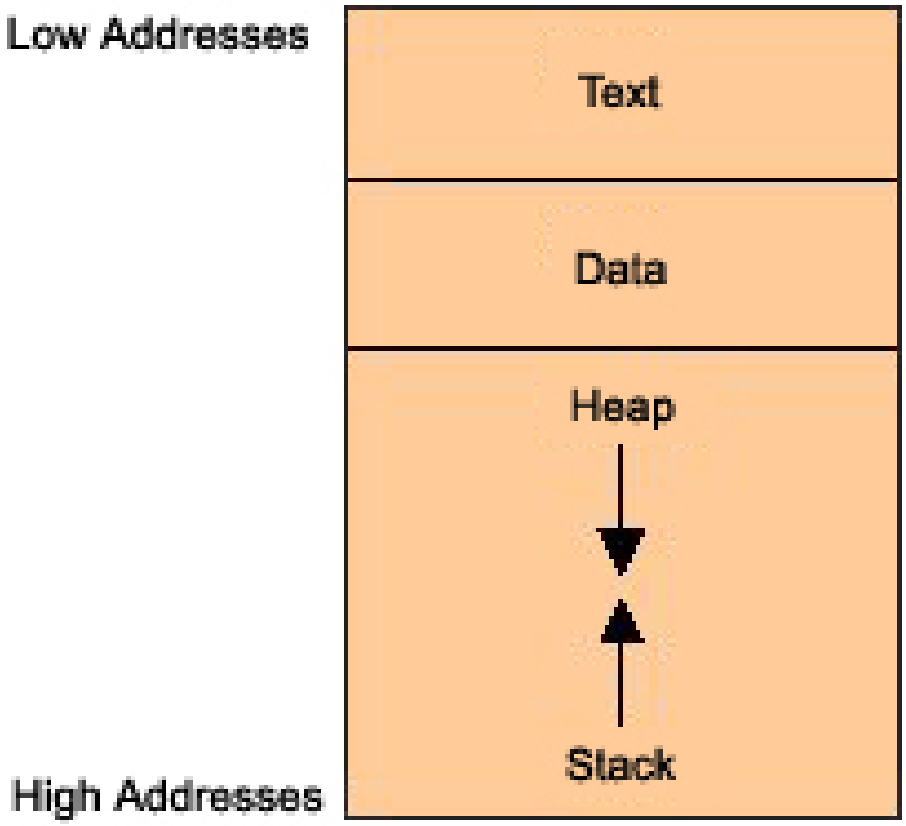


Buffer Overflow

- Text region
  - Fixed by the program
  - Includes code (instructions)
  - Read only
- Data region
  - Contains initialized and uninitialized data
  - Static variables are stored here.
- Stack region
  - LIFO

Buffer Overflow

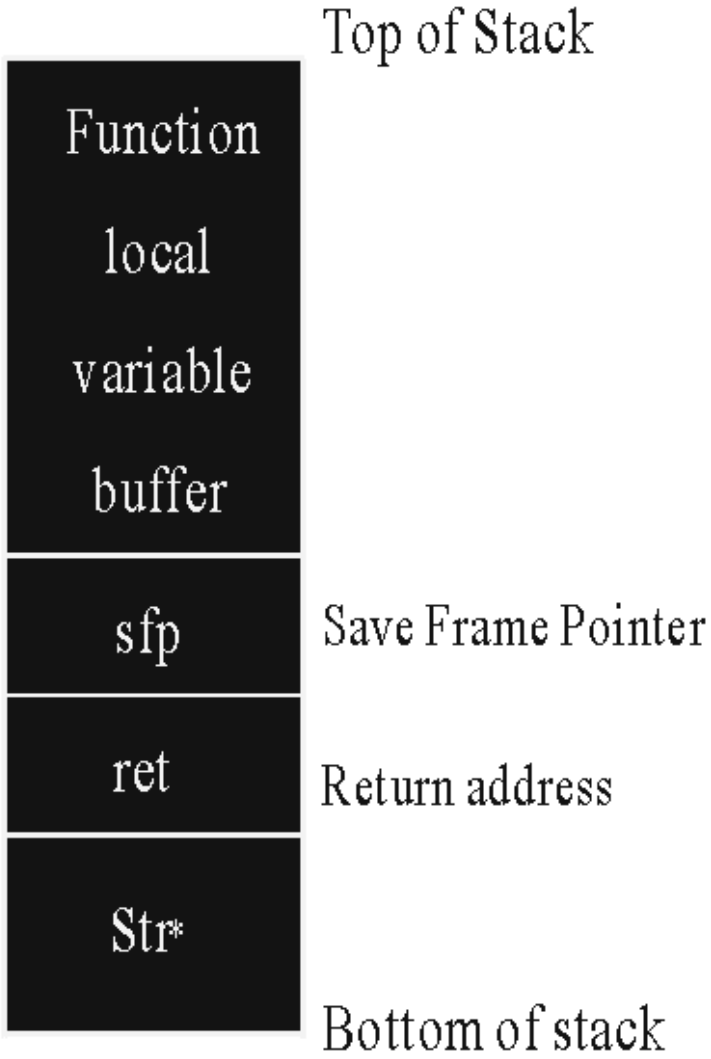
Process Memory Region



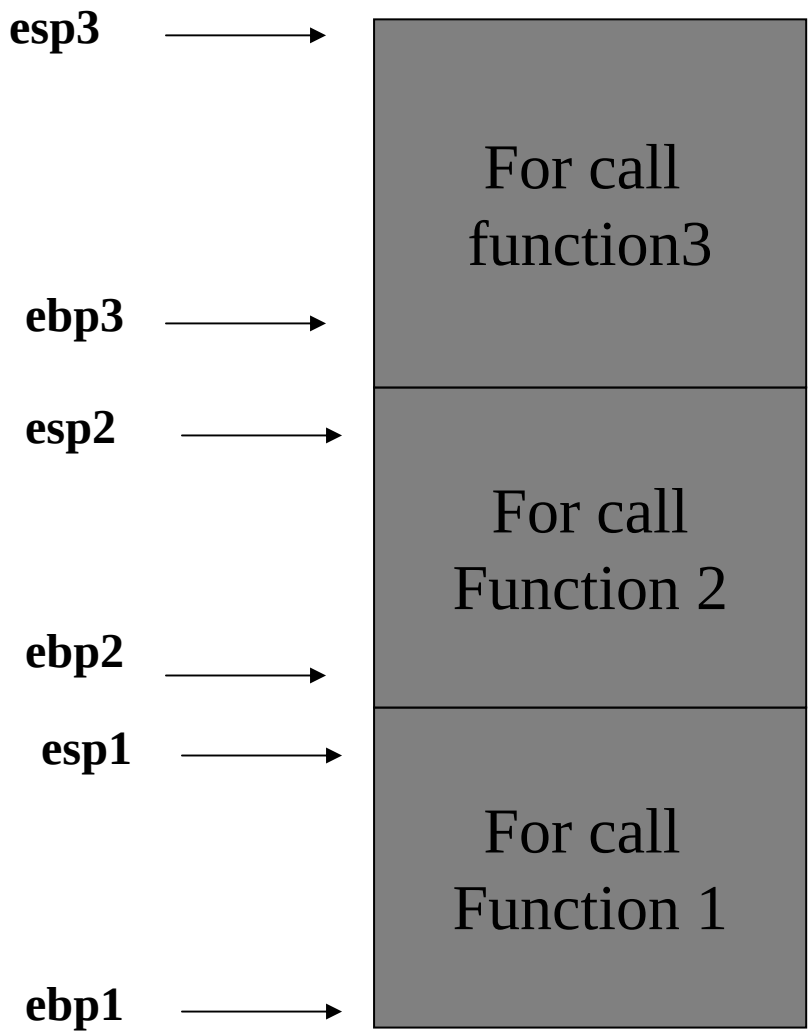
Buffer Overflow

Buffer Overflow Exploit --- An example of buffer overflow program

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}

void main() {
 char large_string[256];
 int i;

 for( i = 0; i < 255; i++)
   large_string[i] = 'A';

 function(large_string);
}
```



Buffer Overflow



esp: stack pointer

ebp: save frame pointer

Disassembly of example1.c

```
Dump of assembler code for function main:
0x80483a0 <main>:       push   %ebp
0x80483a1 <main+1>:     mov    %esp,%ebp
0x80483a3 <main+3>:     push   $0x3
0x80483a5 <main+5>:     push   $0x2
0x80483a7 <main+7>:     push   $0x1
0x80483a9 <main+9>:     call   0x8048398
<function>
0x80483ae <main+14>:    add    $0xc,%esp
0x80483b1 <main+17>:    leave
0x80483b2 <main+18>:    ret
0x80483b3 <main+19>:    nop
0x80483b4 <main+20>:    nop
End of assembler dump.
(gdb) disas function
Dump of assembler code for function function:
0x8048398 <function>:   push   %ebp
0x8048399 <function+1>: mov    %esp,%ebp
0x804839b <function+3>: sub    $0x14,%esp
0x804839e <function+6>: leave
0x804839f <function+7>: ret
End of assembler dump.
```

example1.c source program

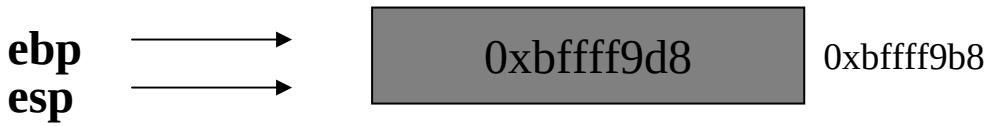
```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```



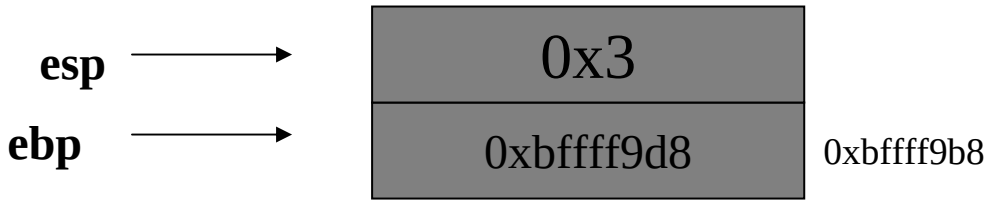
Bottom of the stack

assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```



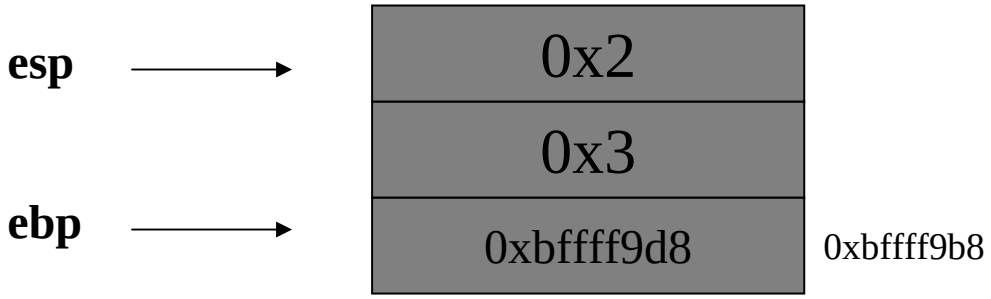
Bottom of the stack

assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

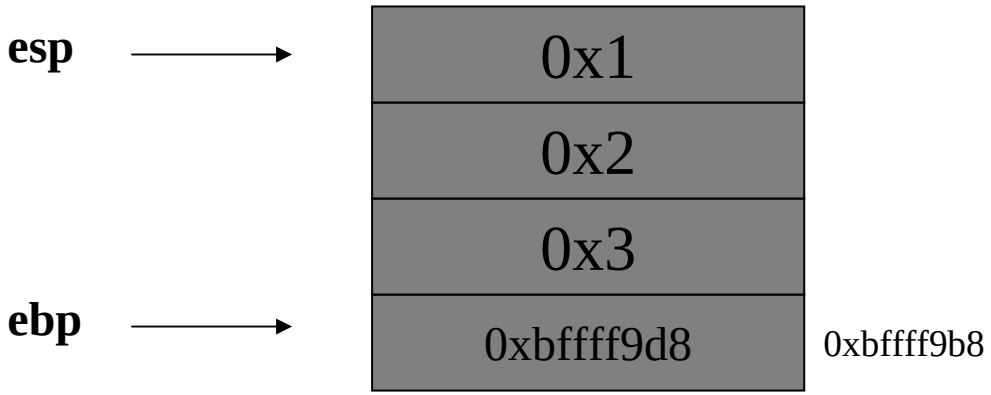


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

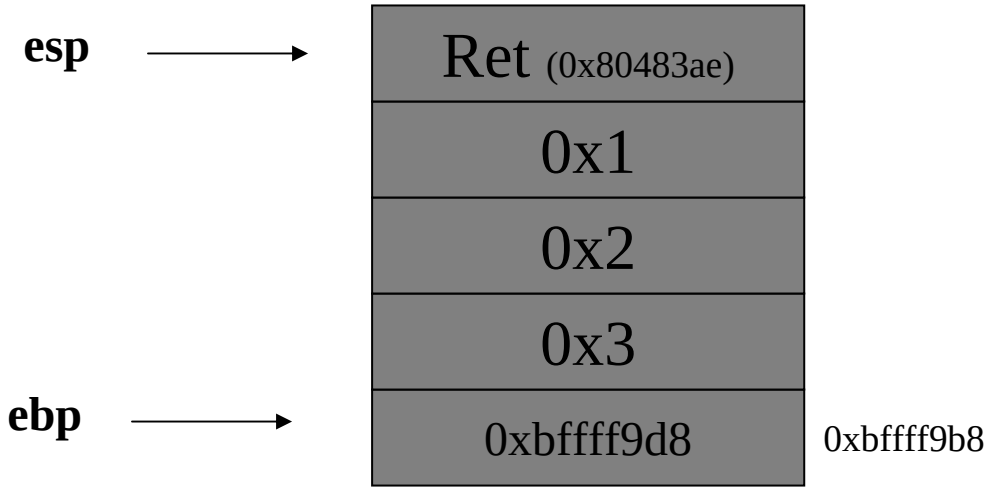


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp      < -- addr 0x80483ae
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

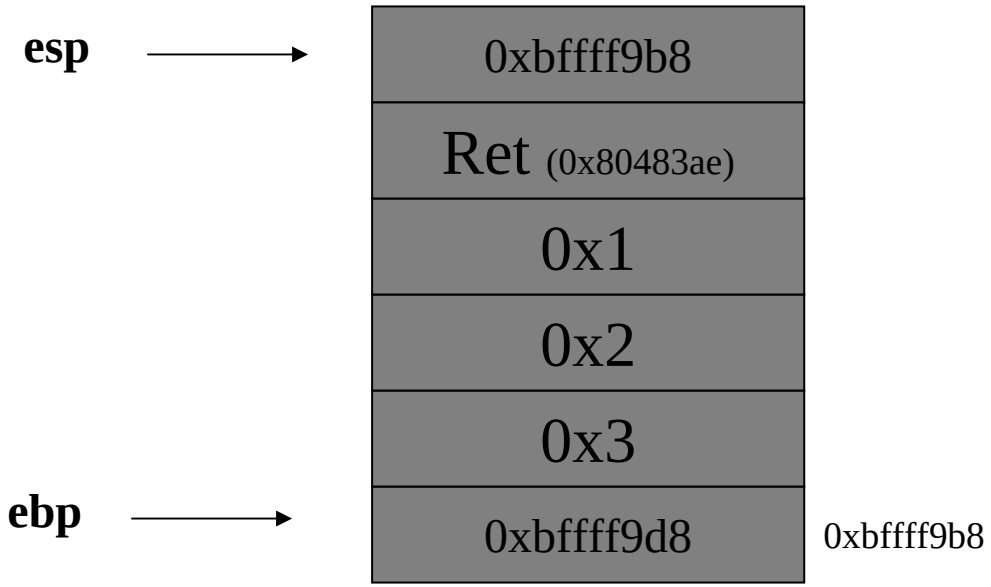


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

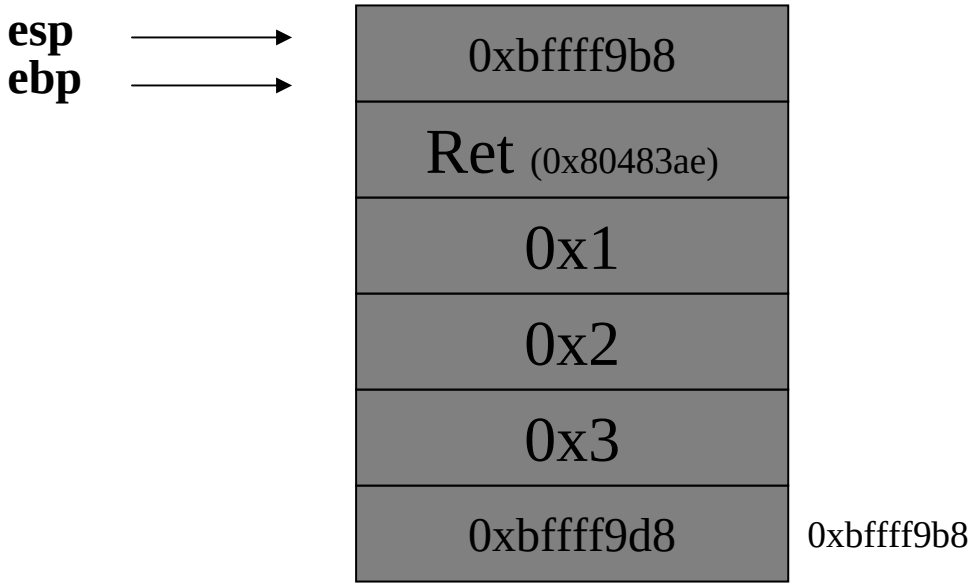


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

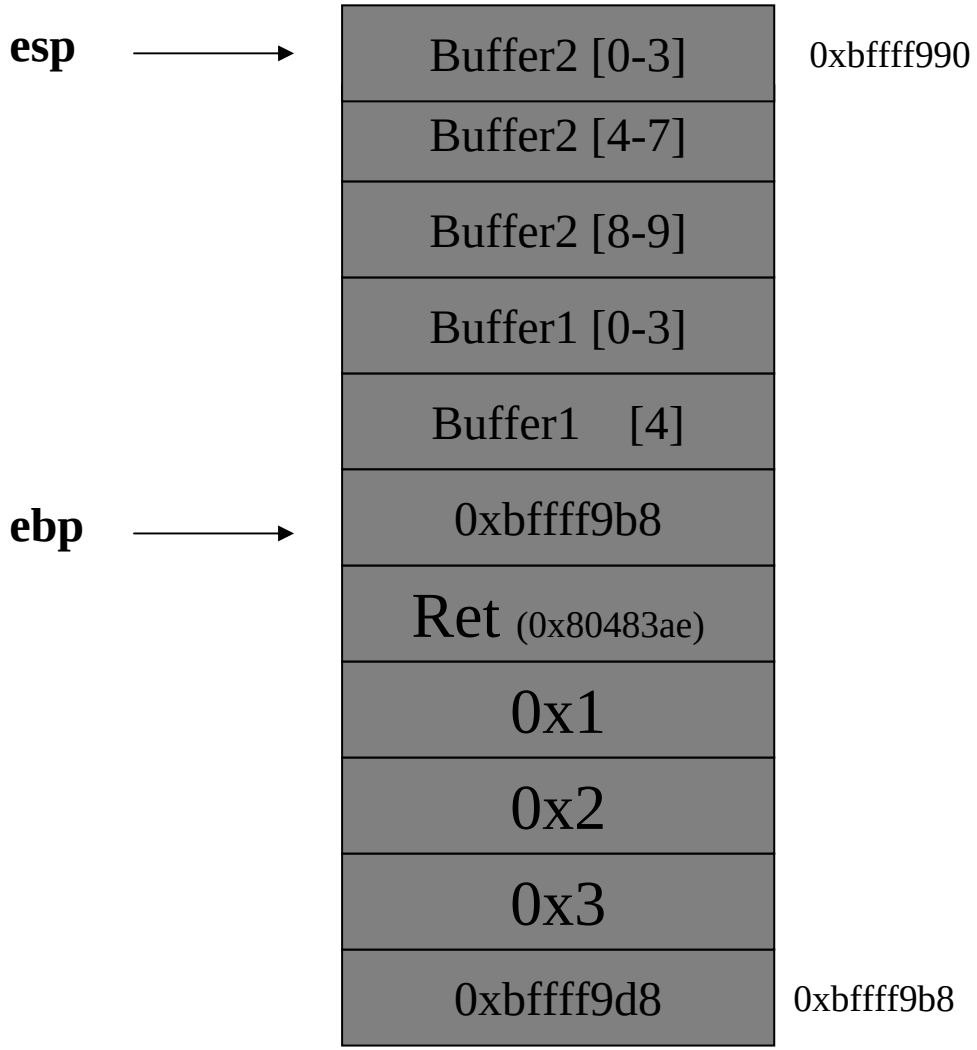


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

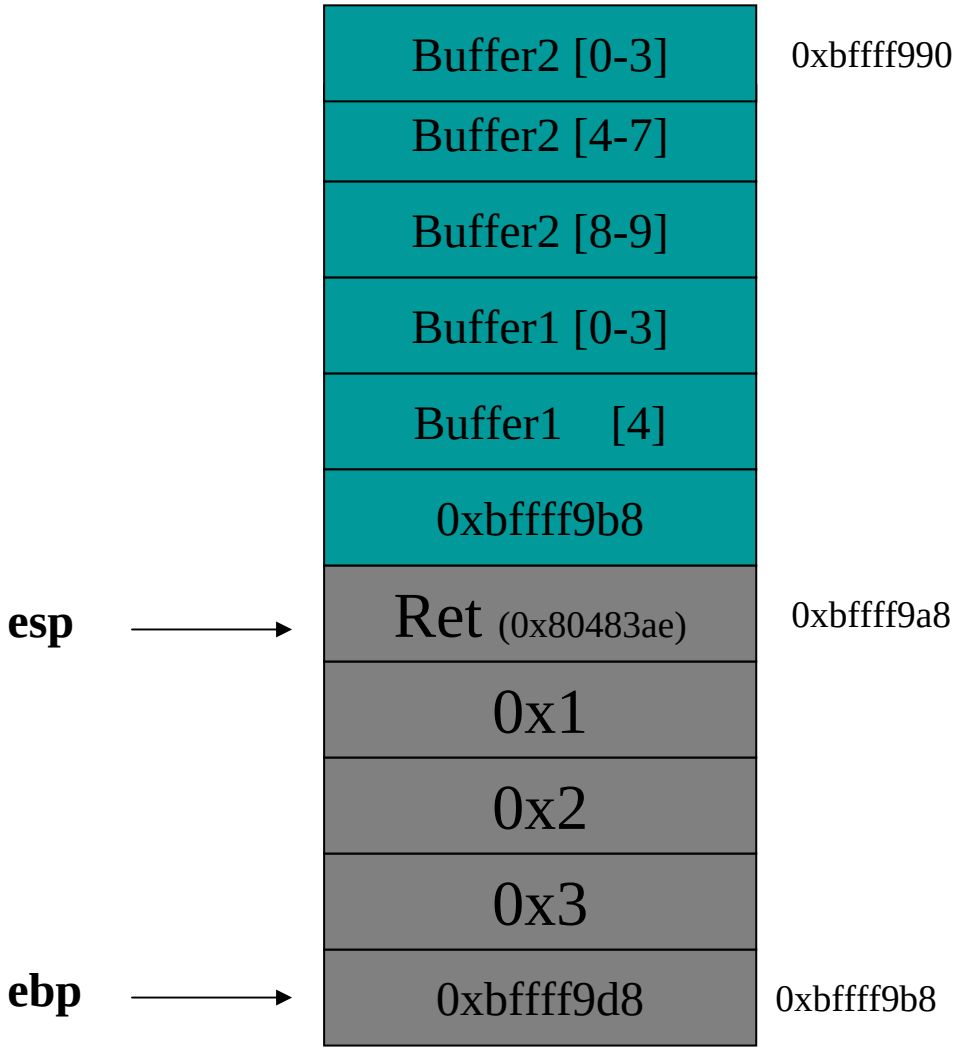


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

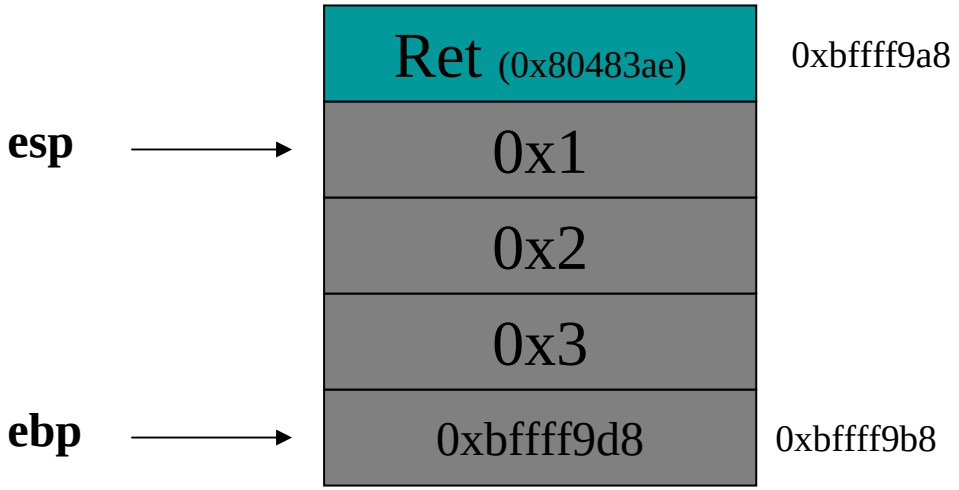


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

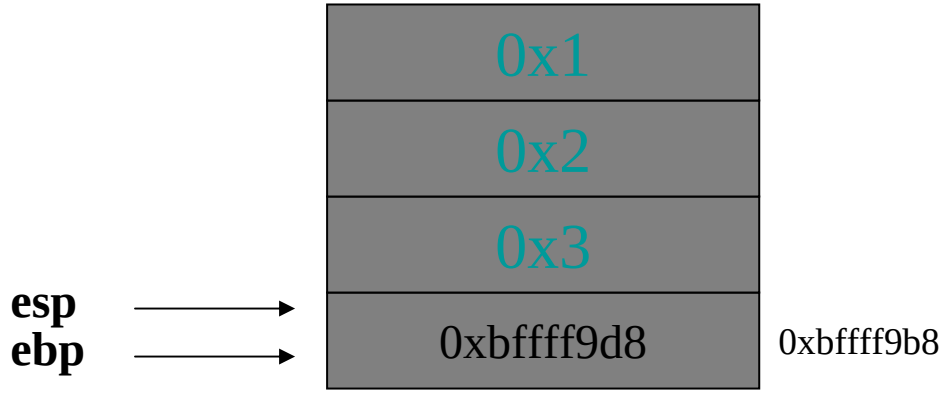


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```

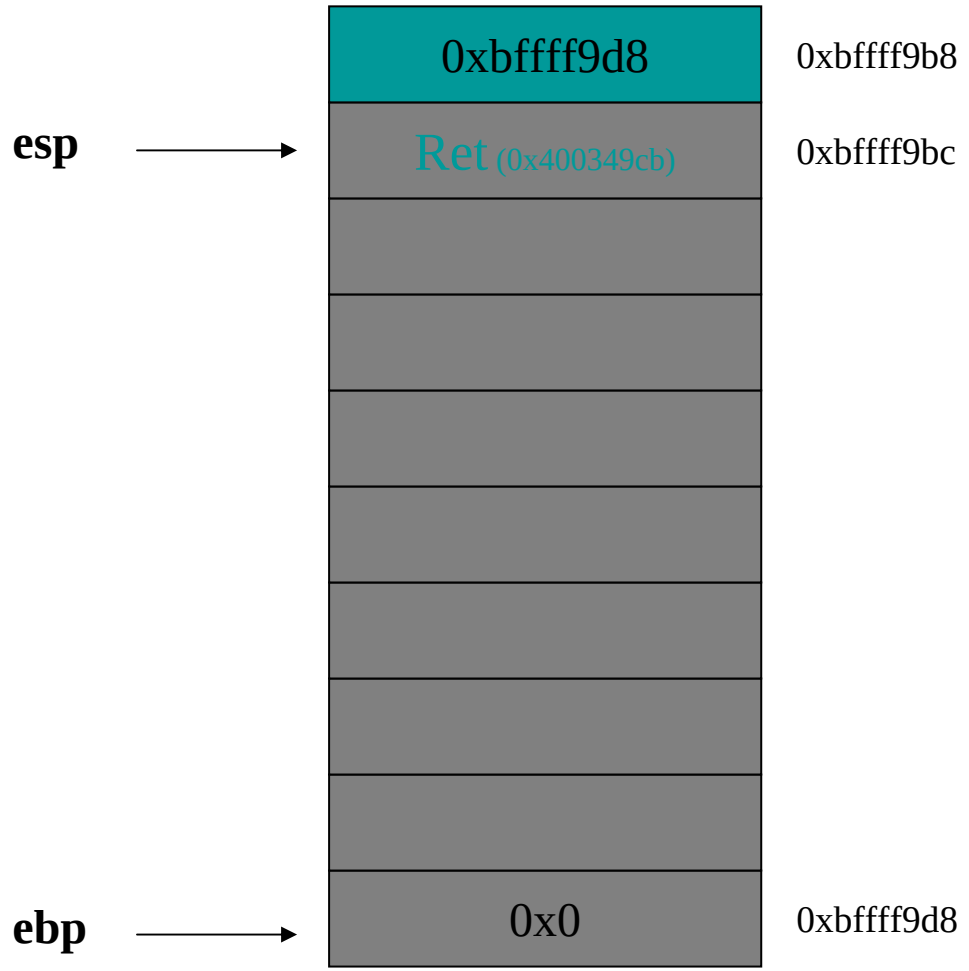


assembler code for main

```
push   %ebp
mov    %esp,%ebp
push   $0x3
push   $0x2
push   $0x1
call   0x8048398 <function>
add    $0xc,%esp
leave
ret
```

assembler code for function

```
push   %ebp
mov    %esp,%ebp
sub    $0x14,%esp
leave
ret
```



Disassembly of example2.c

```
Dump of assembler code for function main:
0x80481b8 <main>:       push   %ebp
0x80481b9 <main+1>:     mov    %esp,%ebp
0x80481bb <main+3>:     sub    $0x104,%esp
0x80481c1 <main+9>:     nop
0x80481c2 <main+10>:    movl   $0x0,0xfffffefc(%ebp)
0x80481cc <main+20>:    lea    0x0(%esi,1),%esi
0x80481d0 <main+24>:    cmpl   $0xfe,0xfffffefc(%ebp)
0x80481da <main+34>:    jle    0x80481e0 <main+40>
0x80481dc <main+36>:    jmp    0x80481f8 <main+64>
0x80481de <main+38>:    mov    %esi,%esi
0x80481e0 <main+40>:    lea    0xffffff00(%ebp),%eax
0x80481e6 <main+46>:    mov    0xfffffefc(%ebp),%edx
0x80481ec <main+52>:    movb   $0x41,(%edx,%eax,1)
0x80481f0 <main+56>:    incl   0xfffffefc(%ebp)
0x80481f6 <main+62>:    jmp    0x80481d0 <main+24>
0x80481f8 <main+64>:    lea    0xffffff00(%ebp),%eax
0x80481fe <main+70>:    push   %eax
0x80481ff <main+71>:    call   0x80481a0 <function>
0x8048204 <main+76>:    add    $0x4,%esp
0x8048207 <main+79>:    leave
0x8048208 <main+80>:    ret
```

example2.c source program

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

Disassembly of example2.c

```
Dump of assembler code for function function:
0x80481a0 <function>:   push   %ebp
0x80481a1 <function+1>: mov    %esp,%ebp
0x80481a3 <function+3>: sub    $0x10,%esp
0x80481a6 <function+6>: mov    0x8(%ebp),%eax
0x80481a9 <function+9>: push   %eax
0x80481aa <function+10>:  lea    0xfffffff0(%ebp),%eax
0x80481ad <function+13>: push   %eax
0x80481ae <function+14>:  call   0x804cf10 <strcpy>
0x80481b3 <function+19>: add    $0x8,%esp
0x80481b6 <function+22>: leave
0x80481b7 <function+23>: ret
End of assembler dump.
```

example2.c source program

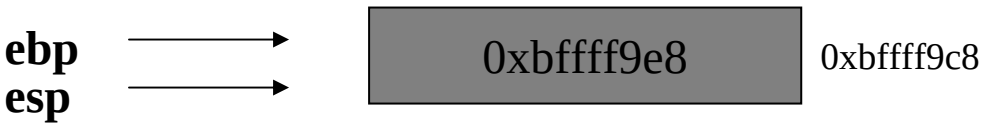
```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x104,%esp
nop
movl   $0x0,0xfffffefc(%ebp)
lea    0x0(%esi,1),%esi
cmpl   $0xfe,0xfffffefc(%ebp)
jle    0x80481e0 <main+40>
jmp    0x80481f8 <main+64>
mov    %esi,%esi
lea    0xffffff00(%ebp),%eax
mov    0xfffffefc(%ebp),%edx
movb   $0x41,(%edx,%eax,1)
incl   0xfffffefc(%ebp)
jmp    0x80481d0 <main+24>
lea    0xffffff00(%ebp),%eax
push   %eax
call   0x80481a0 <function>
add    $0x4,%esp
leave
ret
```

source code of **main**

```
void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

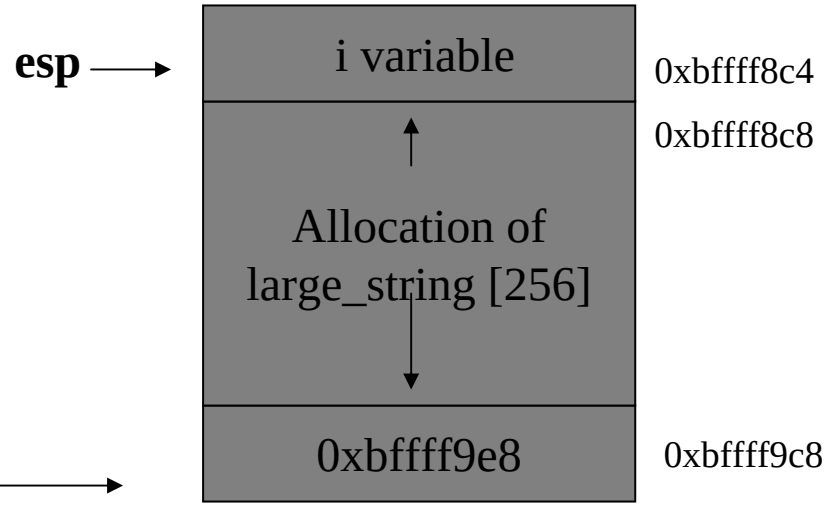


assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x104,%esp
nop
movl   $0x0,0xfffffefc(%ebp)    // initialize counter i variable (top of stack)
lea    0x0(%esi,1),%esi          //initialize source index
cmpl   $0xfe,0xfffffefc(%ebp)    // test if i < 255
jle    0x80481e0 <main+40>       // jump to lea instruction below
jmp    0x80481f8 <main+64>       //exit the for loop
mov    %esi,%esi                 // update the source index
lea    0xffffff00(%ebp),%eax     // load accumulator with data offset
mov    0xfffffefc(%ebp),%edx     // move the i variable to data register
movb   $0x41,(%edx,%eax,1)       //move 'A' char to the data block
incl   0xfffffefc(%ebp)          // i++
jmp    0x80481d0 <main+24>       // loop back to the cmpl instruction above
lea    0xffffff00(%ebp),%eax     // load accumulator with data offset
push   %eax                      // push the large_string pointer to stack
call   0x80481a0 <function>
add    $0x4,%esp
leave
ret
```

source code of **main**

```
void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

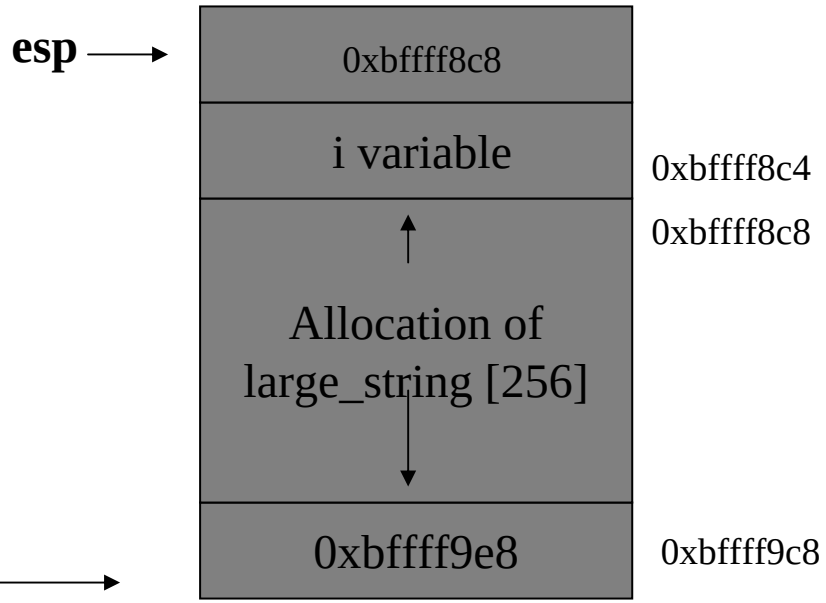


assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x104,%esp
nop
movl   $0x0,0xfffffefc(%ebp)    // initialize counter i variable (top of stack)
lea    0x0(%esi,1),%esi          //initialize source index
cmpl   $0xfe,0xfffffefc(%ebp)    // test if i < 255
jle    0x80481e0 <main+40>       // jump to lea instruction below
jmp    0x80481f8 <main+64>       //exit the for loop
mov    %esi,%esi                 // update the source index
lea    0xffffff00(%ebp),%eax     // load accumulator with data offset
mov    0xfffffefc(%ebp),%edx     // move the i variable to data register
movb   $0x41,(%edx,%eax,1)       //move 'A' char to the data block
incl   0xfffffefc(%ebp)          // i++
jmp    0x80481d0 <main+24>       // loop back to the cmpl instruction above
lea    0xffffff00(%ebp),%eax     // load accumulator with data offset
push   %eax                      // push the large_string pointer to stack
call   0x80481a0 <function>
add    $0x4,%esp
leave
ret
```

source code of **main**

```
void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

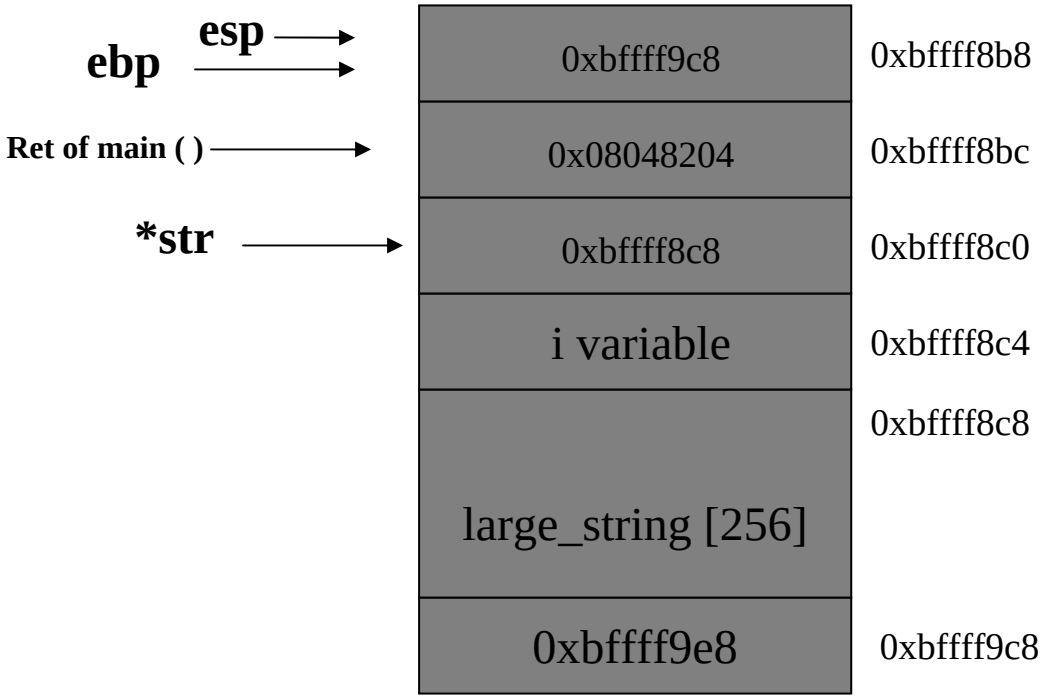


assembler code of function

```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
mov    0x8(%ebp),%eax
push   %eax
lea    0xfffffff0(%ebp),%eax
push   %eax
call   0x804cf10 <strcpy>
add    $0x8,%esp
leave
ret
```

source code of **function**

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}
```

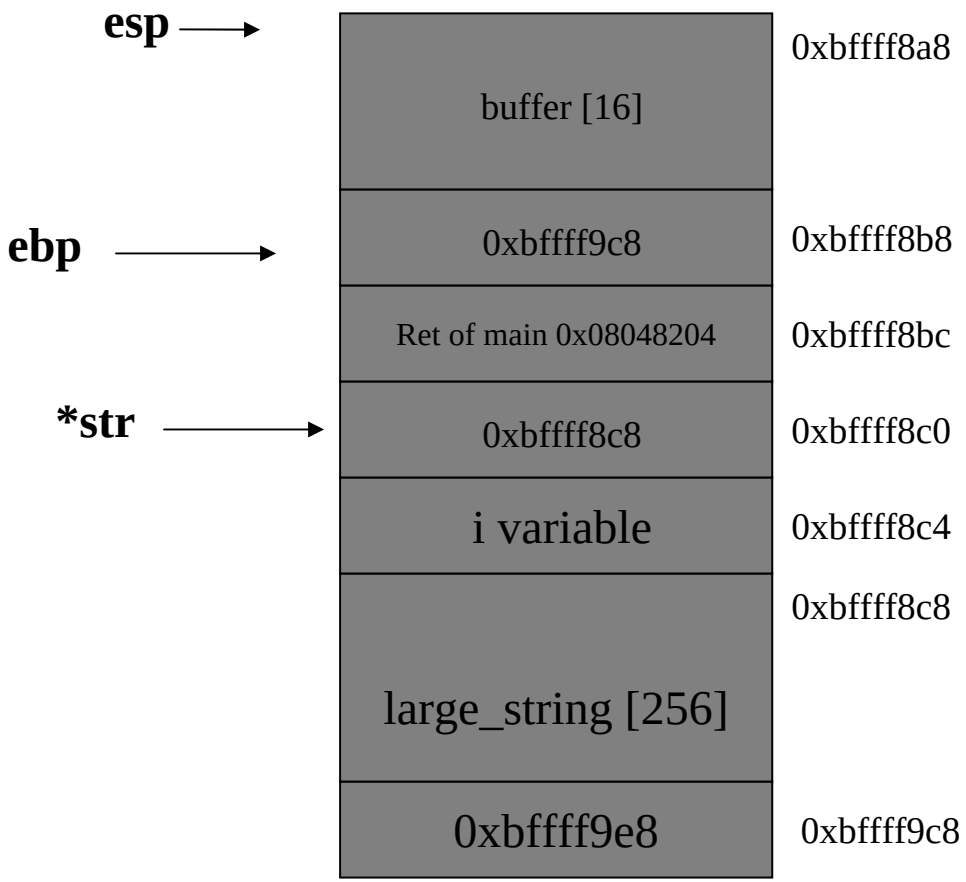


assembler code of function

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
mov     0x8(%ebp),%eax
push    %eax
lea     0xfffffff0(%ebp),%eax
push    %eax
call    0x804cf10 <strcpy>
add     $0x8,%esp
leave
ret
```

source code of **function**

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}
```

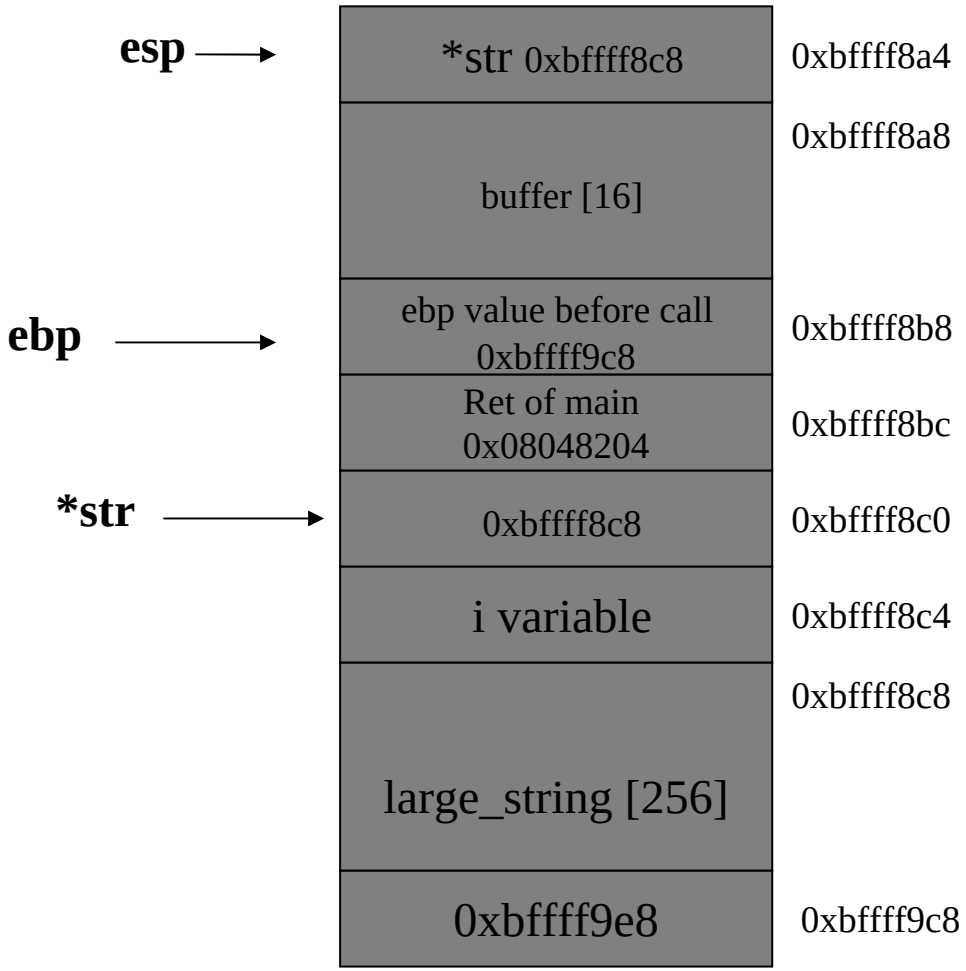


assembler code of function

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
mov     0x8(%ebp),%eax      // update the accumulator pointed to *str
push    %eax
lea     0xfffffff0(%ebp),%eax
push    %eax
call    0x804cf10 <strcpy>
add     $0x8,%esp
leave
ret
```

source code of **function**

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}
```

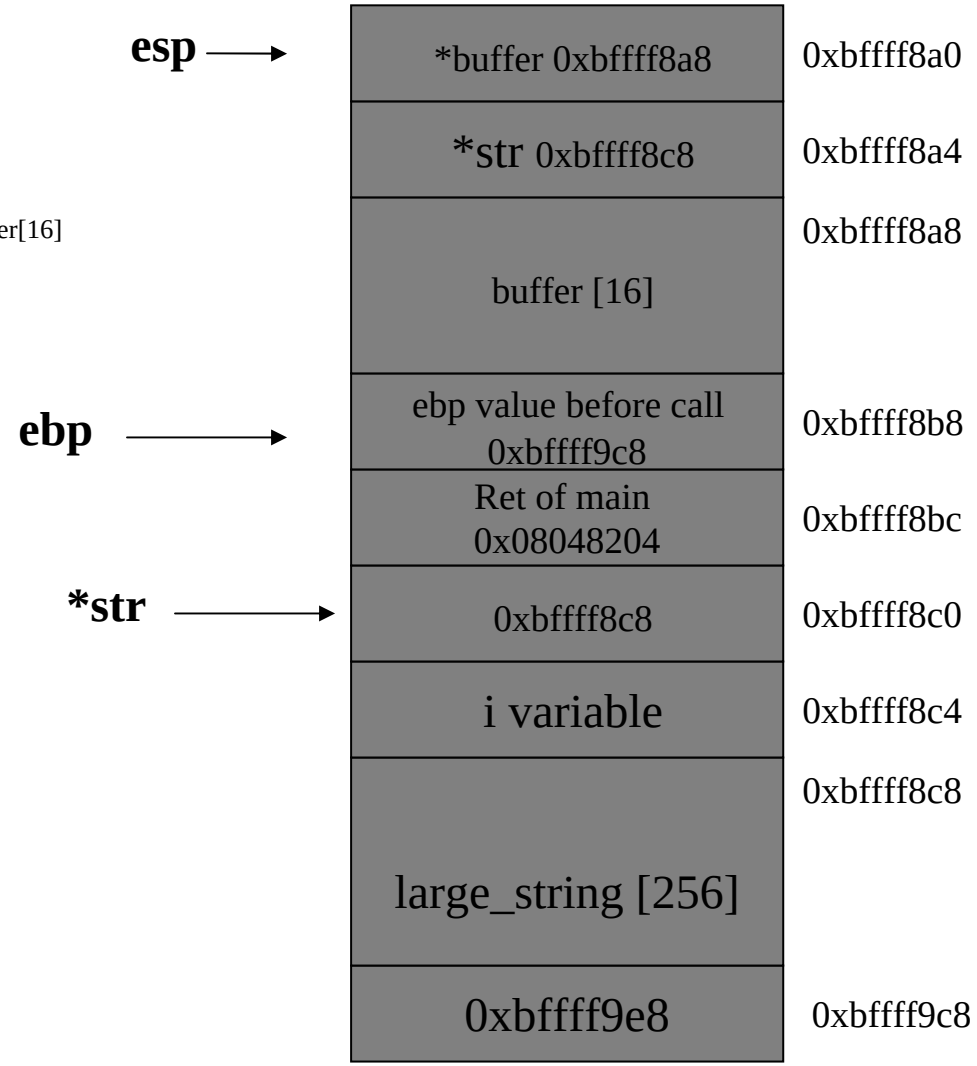


assembler code of function

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
mov     0x8(%ebp),%eax       // update the accumulator pointed to *str
push    %eax
lea     0xfffffff0(%ebp),%eax  // update the accumulator pointed to buffer[16]
push    %eax
call    0x804cf10 <strcpy>
add     $0x8,%esp
leave
ret
```

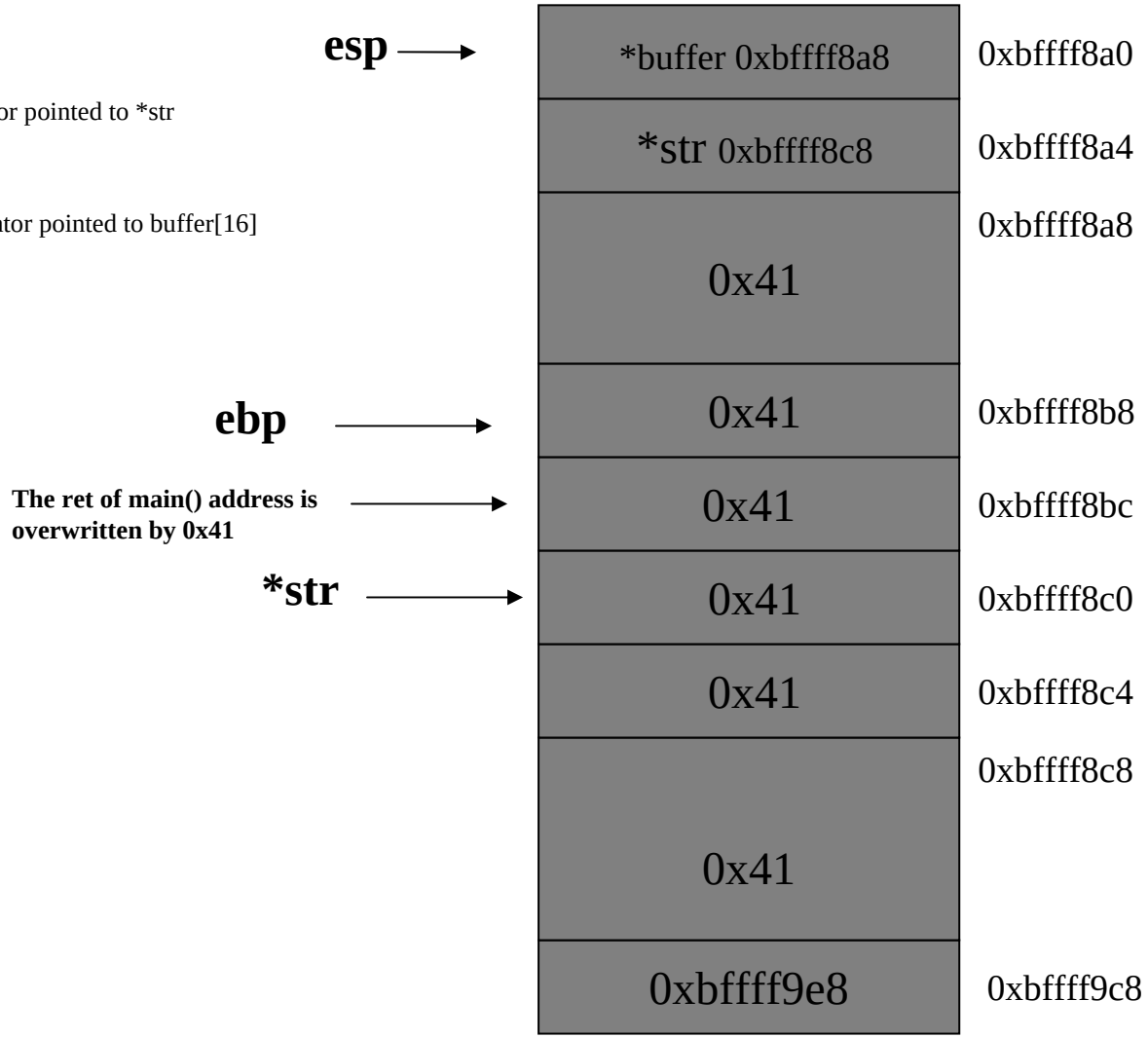
source code of **function**

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}
```



assembler code of function

```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
mov    0x8(%ebp),%eax       // update the accumulator pointed to *str
push   %eax
lea    0xfffffff0(%ebp),%eax  // update the accumulator pointed to buffer[16]
push   %eax
call   0x804cf10 <strcpy>
add    $0x8,%esp
leave
ret
```

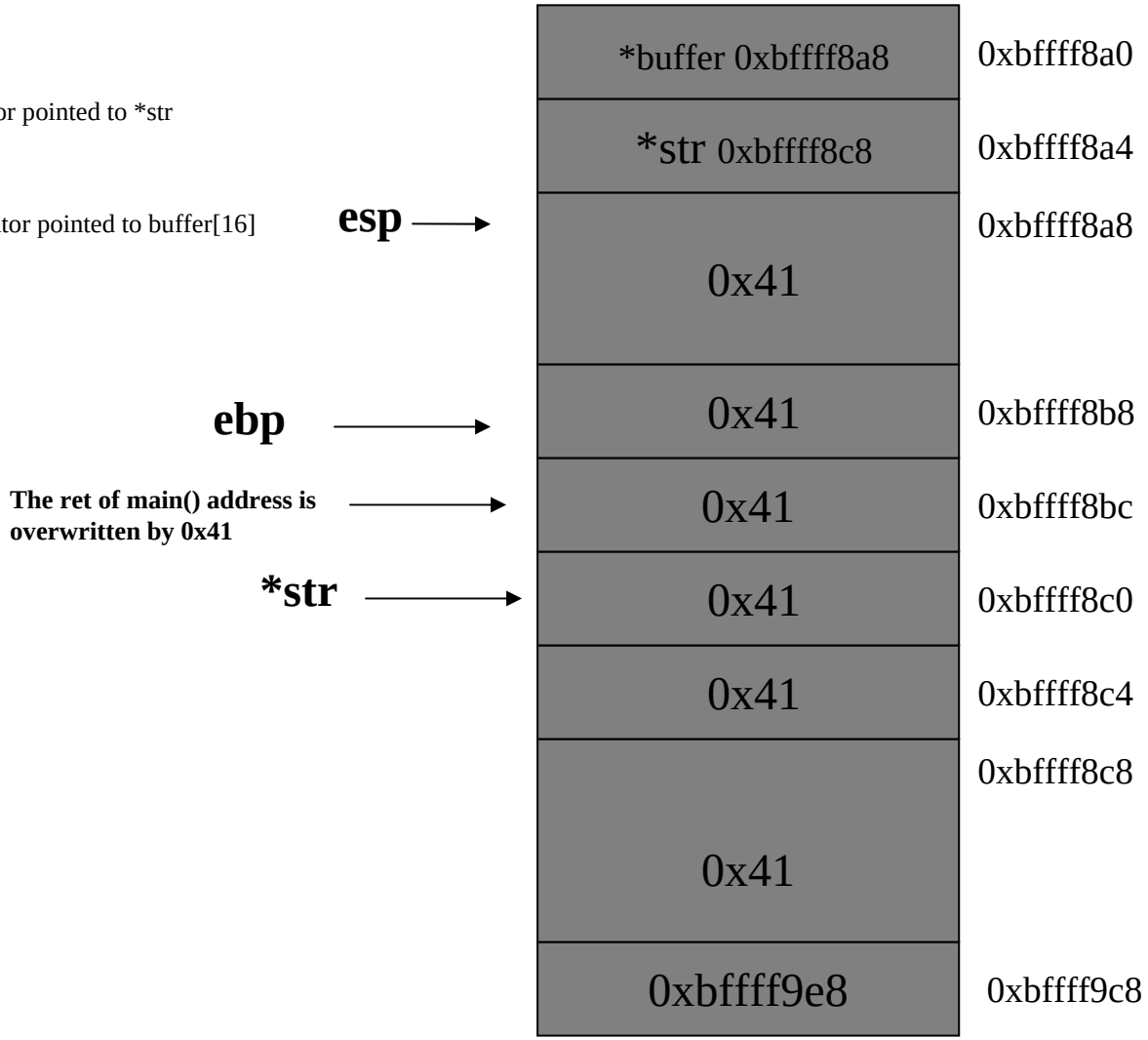


source code of **function**

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}
```

assembler code of function

```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
mov    0x8(%ebp),%eax       // update the accumulator pointed to *str
push   %eax
lea    0xfffffff0(%ebp),%eax  // update the accumulator pointed to buffer[16]
push   %eax
call   0x804cf10 <strcpy>
add    $0x8,%esp
leave
ret
```



source code of **function**

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}
```

assembler code of function

```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
mov    0x8(%ebp),%eax       // update the accumulator pointed to *str
push   %eax
lea    0xfffffff0(%ebp),%eax  // update the accumulator pointed to buffer[16]
push   %eax
call   0x804cf10 <strcpy>
add    $0x8,%esp
leave
ret
```

The ret of main() address is overwritten by 0x41

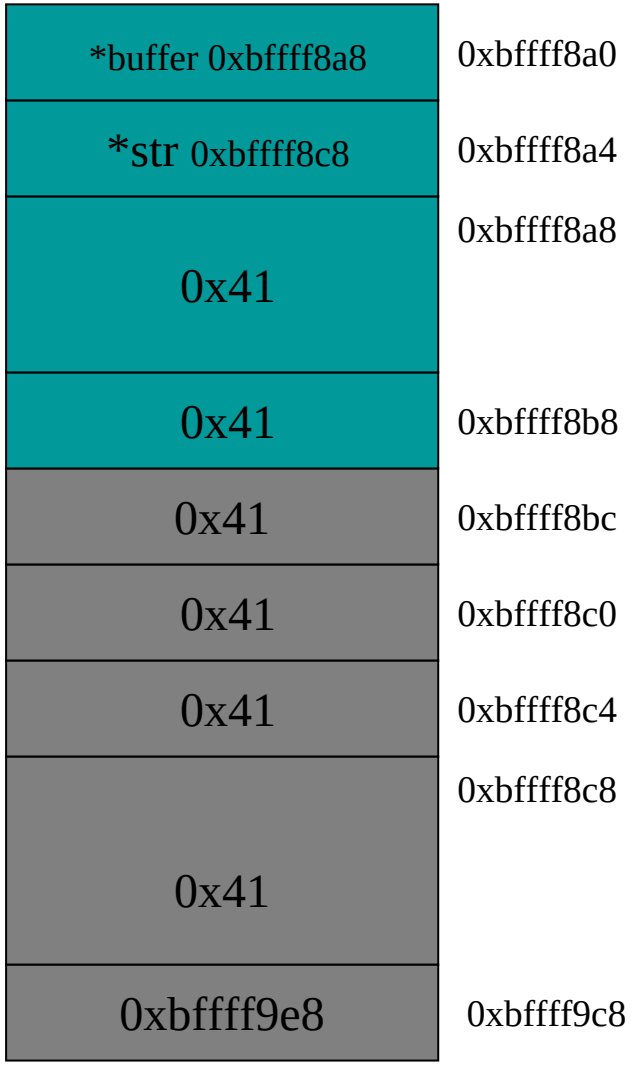
esp →

***str** →

source code of **function**

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}
```

ebp point to 0x41414141



assembler code of function

```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
mov    0x8(%ebp),%eax       // update the accumulator pointed to *str
push   %eax
lea    0xfffffff0(%ebp),%eax  // update the accumulator pointed to buffer[16]
push   %eax
call   0x804cf10 <strcpy>
add    $0x8,%esp
leave
ret
```

The ret of main() address is overwritten by 0x41

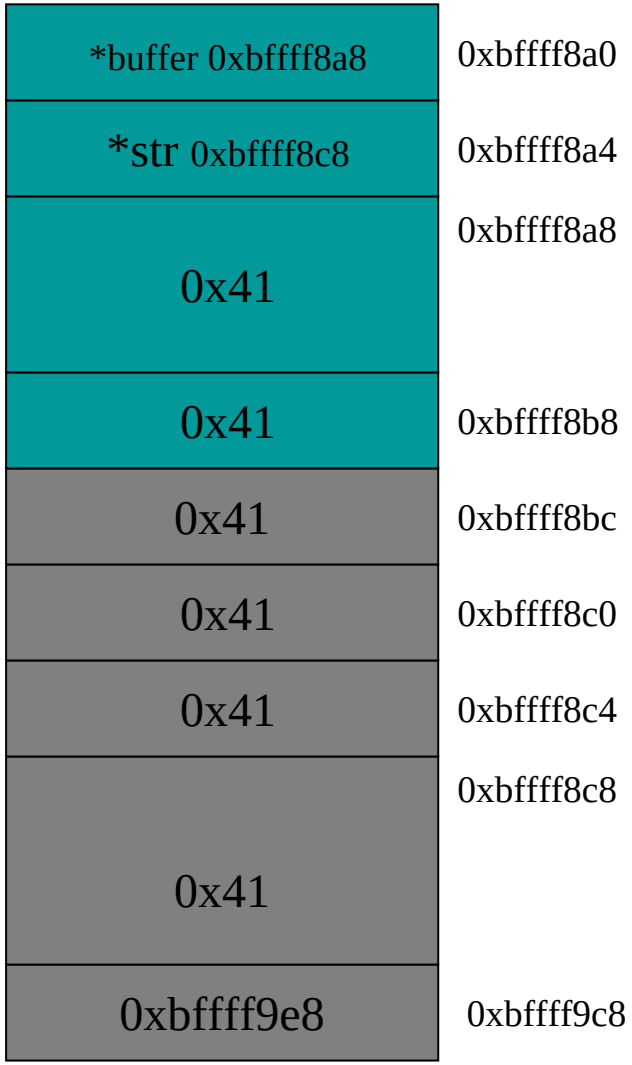
***str**

The return of function leads to segmentation fault

source code of **function**

```
void function(char *str) {
  char buffer[16];

  strcpy(buffer,str);
}
```

ebp point to 0x41414141



Disassembly of example3.c

```
Dump of assembler code for function main:
0x80483e8 <main>:       push   %ebp
0x80483e9 <main+1>:     mov    %esp,%ebp
0x80483eb <main+3>:     sub    $0x4,%esp
0x80483ee <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x80483f5 <main+13>:    push   $0x3
0x80483f7 <main+15>:    push   $0x2
0x80483f9 <main+17>:    push   $0x1
0x80483fb <main+19>:    call   0x80483c8 <function>
0x8048400 <main+24>:    add    $0xc,%esp
0x8048403 <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x804840a <main+34>:    mov    0xfffffffc(%ebp),%eax
0x804840d <main+37>:    push   %eax
0x804840e <main+38>:    push   $0x8048470
0x8048413 <main+43>:    call   0x8048308 <printf>
0x8048418 <main+48>:    add    $0x8,%esp
0x804841b <main+51>:    leave
0x804841c <main+52>:    ret
```

example3.c source program

```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  ret = buffer1 + 12;
  (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Disassembly of example3.c

```
Dump of assembler code for function function:
0x80483c8 <function>:   push   %ebp
0x80483c9 <function+1>: mov    %esp,%ebp
0x80483cb <function+3>: sub    $0x18,%esp
0x80483ce <function+6>: lea    0xfffffff8(%ebp),%eax
0x80483d1 <function+9>: lea    0xc(%eax),%ecx
0x80483d4 <function+12>:        mov    %ecx,0xfffffff8(%ebp)
0x80483d7 <function+15>:        mov    0xfffffff8(%ebp),%eax
0x80483da <function+18>:        mov    0xfffffff8(%ebp),%edx
0x80483dd <function+21>:        mov    (%edx),%ecx
0x80483df <function+23>:        add    $0x8,%ecx
0x80483e2 <function+26>:        mov    %ecx,(%eax)
0x80483e4 <function+28>:        leave
0x80483e5 <function+29>:        ret
```

example3.c source program

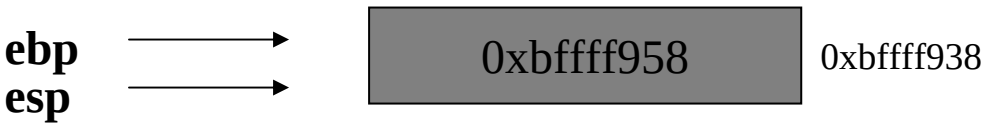
```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;

  ret = buffer1 + 12;
  (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp
movl   $0x0,0xfffffffc(%ebp)
push   $0x3
push   $0x2
push   $0x1
call   0x80483c8 <function>
add    $0xc,%esp
movl   $0x1,0xfffffffc(%ebp)
mov    0xfffffffc(%ebp),%eax
push   %eax
push   $0x8048470
call   0x8048308 <printf>
add    $0x8,%esp
leave
ret
```

source code of **main**

```
void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

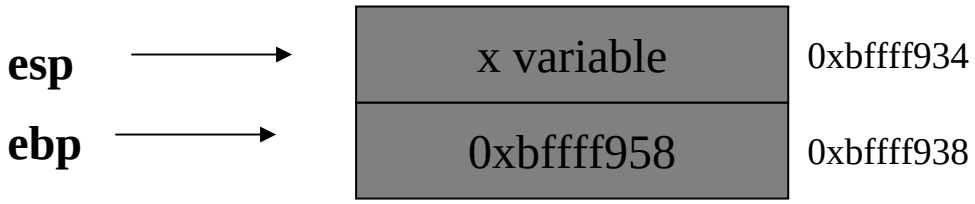


assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp
movl   $0x0,0xfffffffc(%ebp)
push   $0x3
push   $0x2
push   $0x1
call   0x80483c8 <function>
add    $0xc,%esp
movl   $0x1,0xfffffffc(%ebp)
mov    0xfffffffc(%ebp),%eax
push   %eax
push   $0x8048470
call   0x8048308 <printf>
add    $0x8,%esp
leave
ret
```

source code of **main**

```
void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

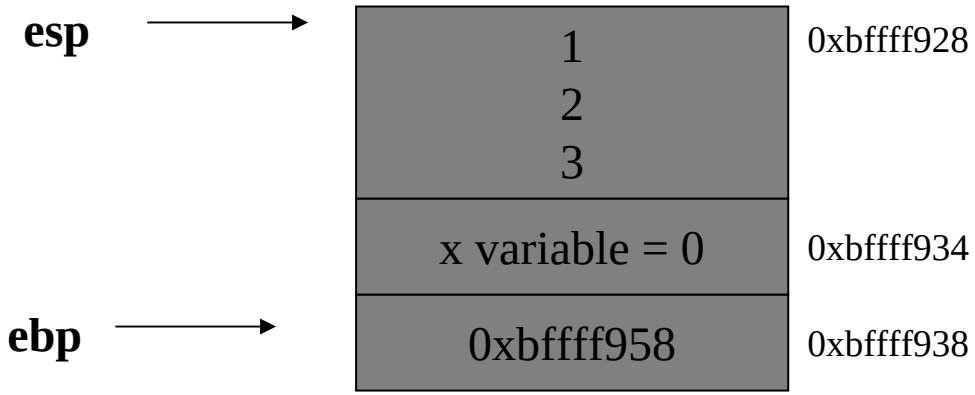


assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp        // allocate x variable in the stack
movl   $0x0,0xfffffffc(%ebp)  // initialize x variable
push   $0x3     // passing arguments to function via stack
push   $0x2
push   $0x1
call   0x80483c8 <function>
add    $0xc,%esp
movl   $0x1,0xfffffffc(%ebp)
mov    0xfffffffc(%ebp),%eax
push   %eax
push   $0x8048470
call   0x8048308 <printf>
add    $0x8,%esp
leave
ret
```

source code of **main**

```
void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```



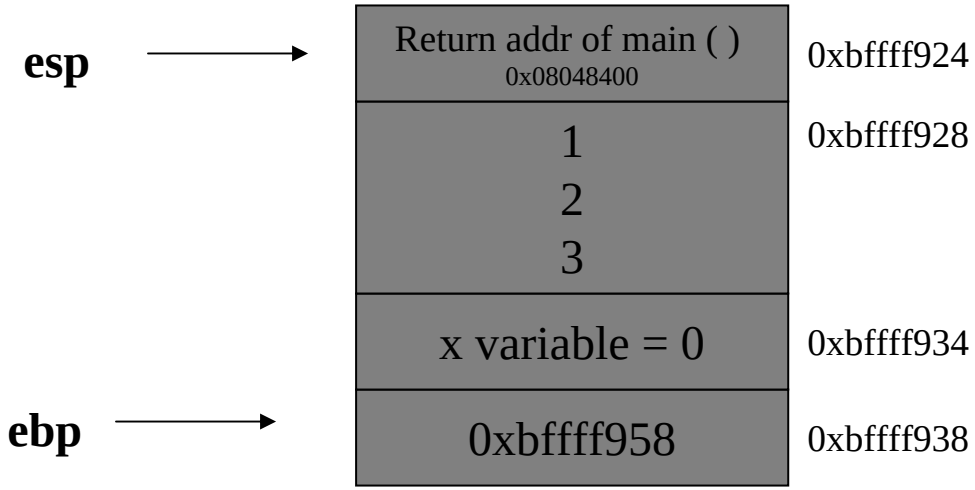
Bottom of the stack

assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp        // allocate x variable in the stack
movl   $0x0,0xfffffffc(%ebp)  // initialize x variable
push   $0x3     // passing arguments to function via stack
push   $0x2
push   $0x1
call   0x80483c8 <function>
add    $0xc,%esp
movl   $0x1,0xfffffffc(%ebp)
mov    0xfffffffc(%ebp),%eax
push   %eax
push   $0x8048470
call   0x8048308 <printf>
add    $0x8,%esp
leave
ret
```

source code of **main**

```
void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

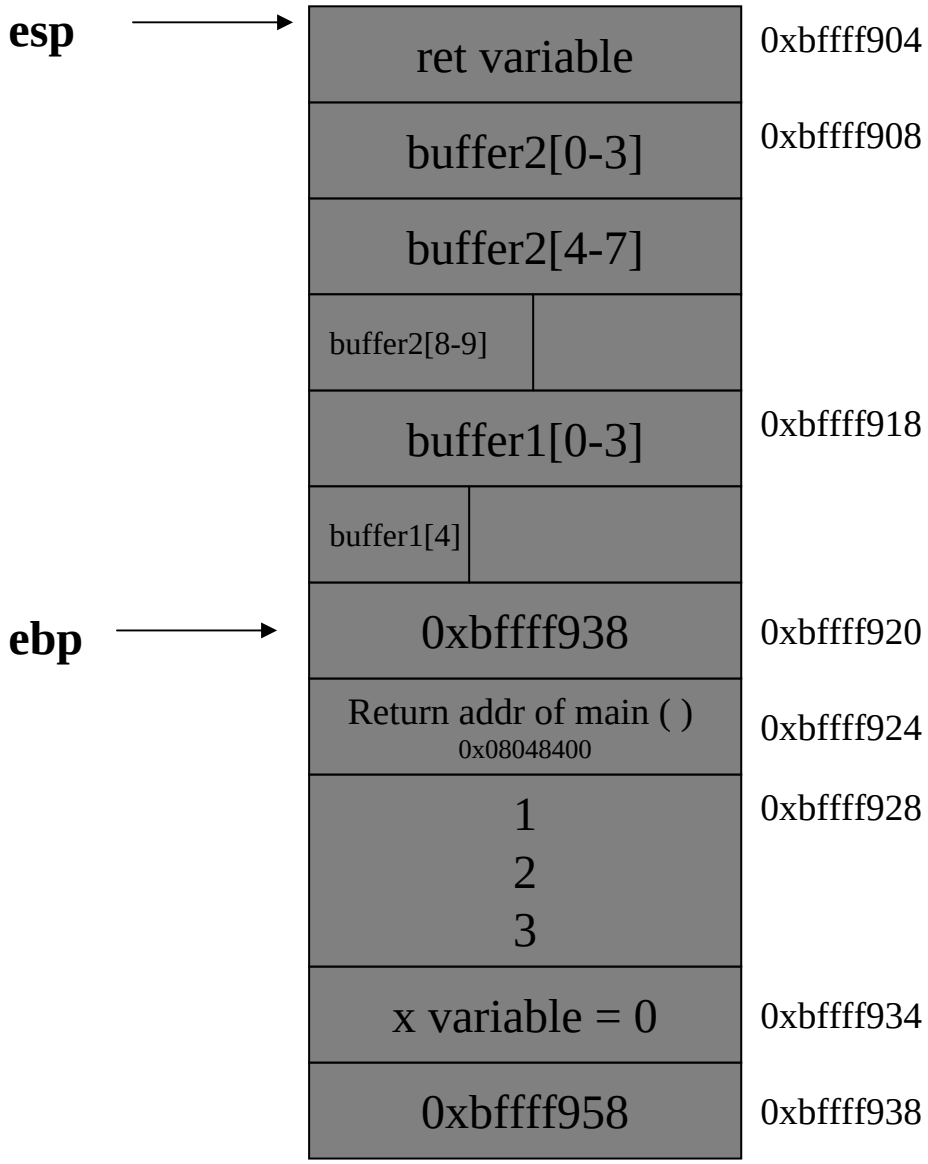


assembler code of **functioin**

```
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp     // allocate buffer1,2 and ret variable
lea    0xffffffff8(%ebp),%eax
lea    0xc(%eax),%ecx
mov    %ecx,0xffffffe8(%ebp)
mov    0xffffffe8(%ebp),%eax
mov    0xffffffe8(%ebp),%edx
mov    (%edx),%ecx
add    $0x8,%ecx
mov    %ecx,(%eax)
leave
ret
```

source code of **function**

```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;
  ret = buffer1 + 12;
  (*ret) += 8;
}
```

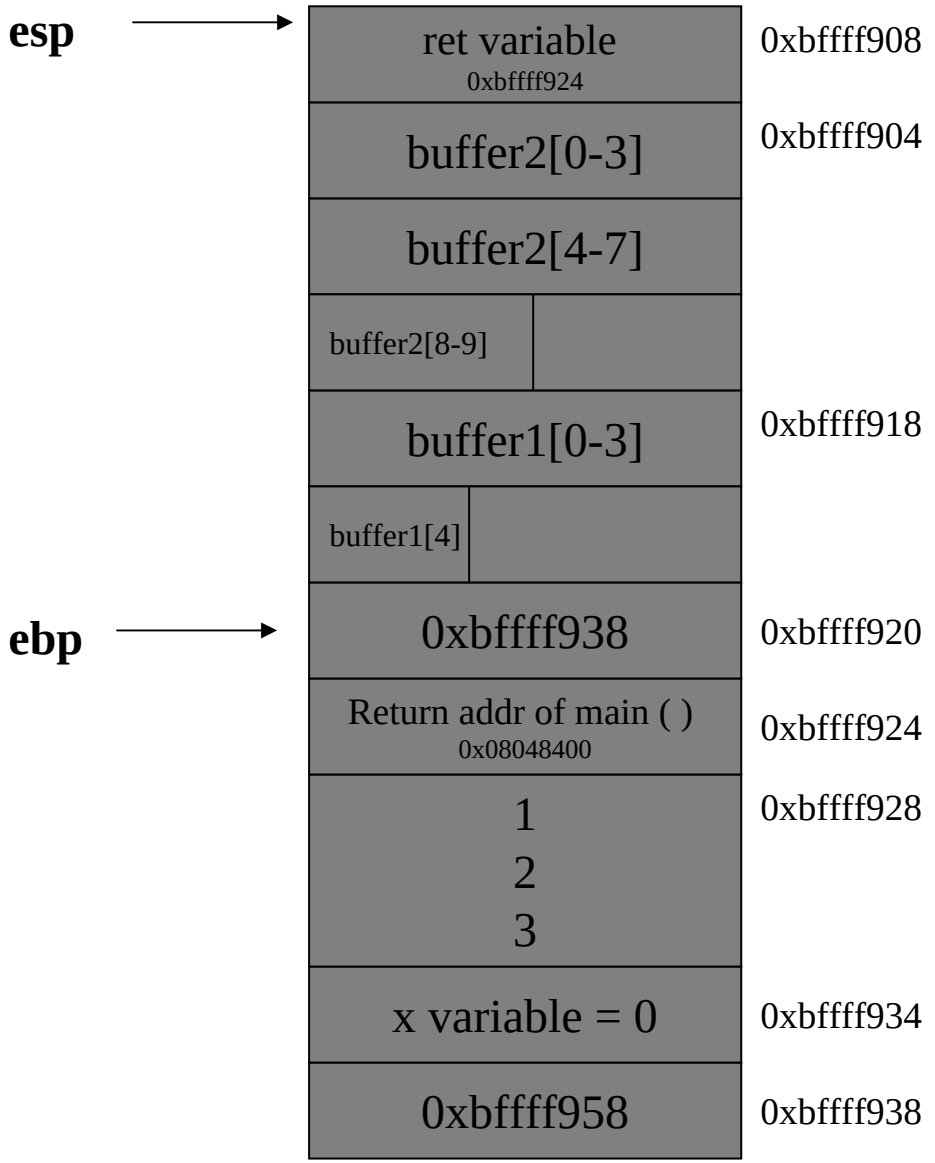


assembler code of **functioin**

```
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp     // allocate buffer1,2 and ret variable
lea    0xffffffff8(%ebp),%eax // make ax points to buffer1
lea    0xc(%eax),%ecx      // make cx points be buffer1 + 12,
                           i.e stack location store the return
                           address of main ( )
mov    %ecx,0xffffffe8(%ebp)  //store the cx result to ret variable
mov    0xffffffe8(%ebp),%eax  //load the ret variable to ax
mov    0xffffffe8(%ebp),%edx  //load the ret variable to dx
mov    (%edx),%ecx      // load the return address of main ( ) to cx
add    $0x8,%ecx        // add 8 to cx
mov    %ecx,(%eax)      // overwrite the return address of main ( )
leave
ret
```

source code of **function**

```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;
  ret = buffer1 + 12;
  (*ret) += 8;
}
```

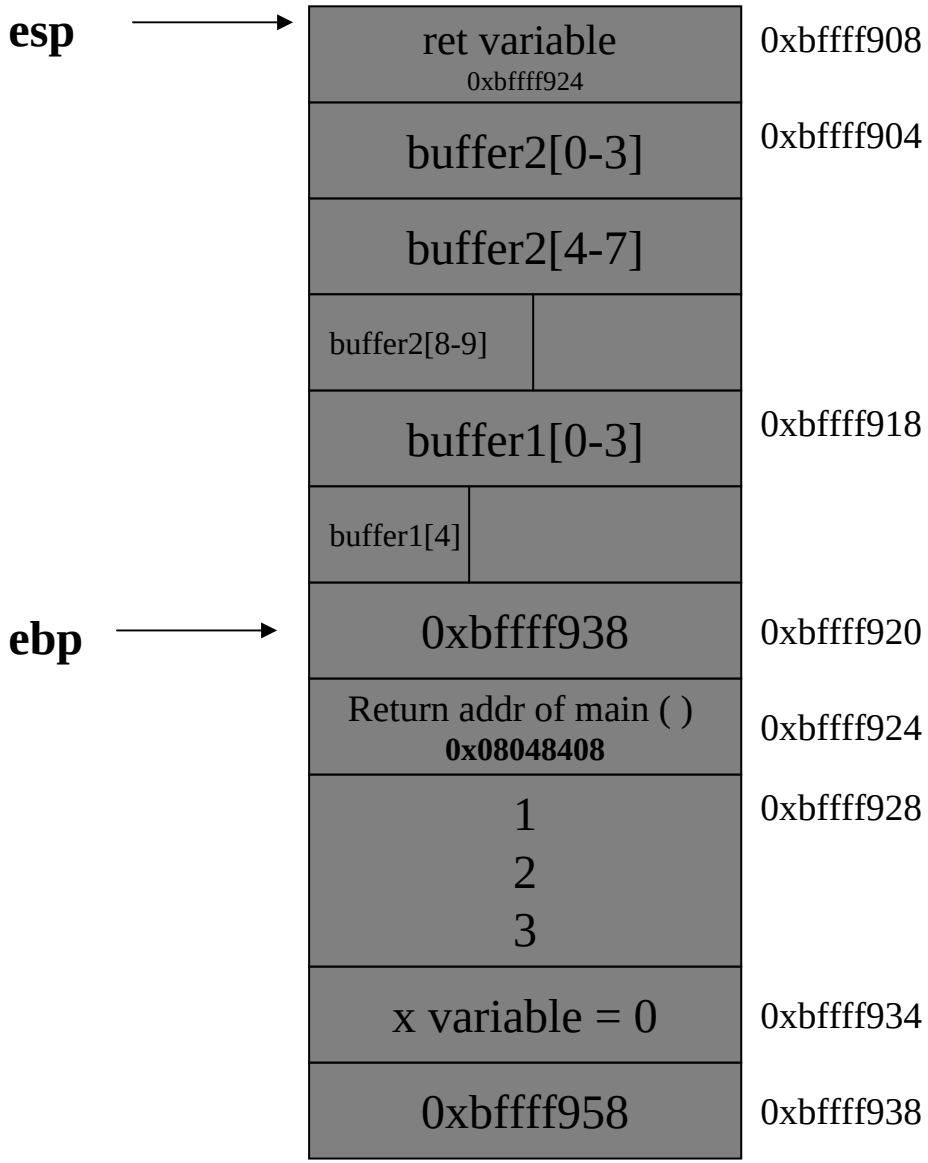


assembler code of **functioin**

```
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp     // allocate buffer1,2 and ret variable
lea    0xffffffff8(%ebp),%eax // make ax points to buffer1
lea    0xc(%eax),%ecx      // make cx points be buffer1 + 12,
                           i.e stack location store the return
                           address of main ( )
mov    %ecx,0xffffffe8(%ebp)  //store the cx result to ret variable
mov    0xffffffe8(%ebp),%eax  //load the ret variable to ax
mov    0xffffffe8(%ebp),%edx  //load the ret variable to dx
mov    (%edx),%ecx      // load the return address of main ( ) to cx
add    $0x8,%ecx        // add 8 to cx
mov    %ecx,(%eax)      // overwrite the return address of main ( )
leave
ret
```

source code of **function**

```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
  int *ret;
  ret = buffer1 + 12;
  (*ret) += 8;
}
```



assembler code of **main**

```
0x80483e8 <main>:      push   %ebp
0x80483e9 <main+1>:    mov    %esp,%ebp
0x80483eb <main+3>:    sub    $0x4,%esp
0x80483ee <main+6>:    movl   $0x0,0xfffffffc(%ebp)
0x80483f5 <main+13>:   push   $0x3
0x80483f7 <main+15>:   push   $0x2
0x80483f9 <main+17>:   push   $0x1
0x80483fb <main+19>:   call   0x80483c8 <function>
0x8048400 <main+24>:   add    $0xc,%esp          // original return point after call, adjust the stack pointer for the passing arguments
0x8048403 <main+27>:   movl   $0x1,0xfffffffc(%ebp)  // assign 1 to variable x
0x804840a <main+34>:   mov    0xfffffffc(%ebp),%eax  // the actual return point after call, load the variable x to ax
0x804840d <main+37>:   push   %eax                   // pass the value x to printf via stack
0x804840e <main+38>:   push   $0x8048470
0x8048413 <main+43>:   call   0x8048308 <printf>
0x8048418 <main+48>:   add    $0x8,%esp        // adjust the stack pointer for passing arguments
0x804841b <main+51>:   leave
0x804841c <main+52>:   ret
```

Disassembly of testsc.c

```
Dump of assembler code for function main:
0x8048398 <main>:       push   %ebp
0x8048399 <main+1>:     mov    %esp,%ebp
0x804839b <main+3>:     sub    $0x4,%esp
0x804839e <main+6>:     lea    0xfffffffc(%ebp),%eax
0x80483a1 <main+9>:     lea    0x8(%eax),%edx
0x80483a4 <main+12>:    mov    %edx,0xfffffffc(%ebp)
0x80483a7 <main+15>:    mov    0xfffffffc(%ebp),%eax
0x80483aa <main+18>:    movl   $0x8049440,(%eax)
0x80483b0 <main+24>:    leave
0x80483b1 <main+25>:    ret
0x80483b2 <main+26>:    nop
```

testsc.c source program

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";


void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

Disassembly of shellcode

```
(gdb) disas 0x8049440
Dump of assembler code for function shellcode:
0x8049440 <shellcode>:  jmp    0x804946c <shellcode+44>
0x8049442 <shellcode+2>:        pop    %esi
0x8049443 <shellcode+3>:        mov    %esi,0x8(%esi)
0x8049446 <shellcode+6>:        movb   $0x0,0x7(%esi)
0x804944a <shellcode+10>:       movl   $0x0,0xc(%esi)
0x8049451 <shellcode+17>:       mov    $0xb,%eax
0x8049456 <shellcode+22>:       mov    %esi,%ebx
0x8049458 <shellcode+24>:       lea    0x8(%esi),%ecx
0x804945b <shellcode+27>:       lea    0xc(%esi),%edx
0x804945e <shellcode+30>:       int    $0x80
0x8049460 <shellcode+32>:       mov    $0x1,%eax
0x8049465 <shellcode+37>:       mov    $0x0,%ebx
0x804946a <shellcode+42>:       int    $0x80
0x804946c <shellcode+44>:       call   0x8049442 <shellcode+2>
0x8049471 <shellcode+49>:       das
0x8049472 <shellcode+50>:       bound  %ebp,0x6e(%ecx)
0x8049475 <shellcode+53>:       das
0x8049476 <shellcode+54>:       jae    0x80494e0 <_DYNAMIC+48>
0x8049478 <shellcode+56>:       add    %cl,0xc35dec(%ecx)
0x804947e <shellcode+62>:       add    %al,(%eax)
```

testsc.c source program

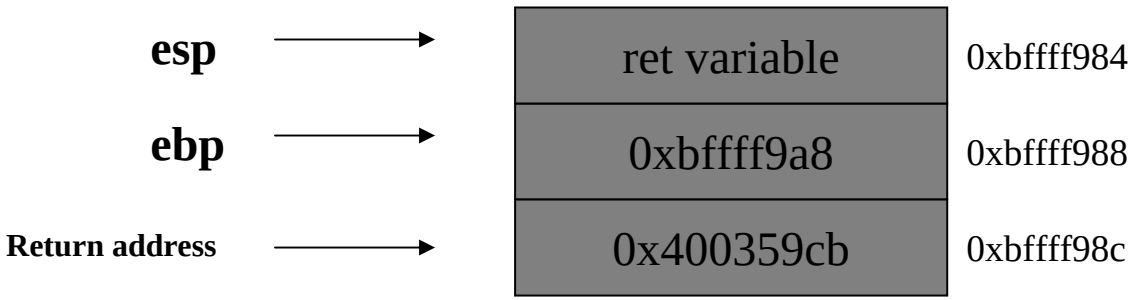
```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\
x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\
x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\
x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x
5d\xc3";


void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;


}
```

assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp
lea    0xfffffffc(%ebp),%eax
lea    0x8(%eax),%edx
mov    %edx,0xfffffffc(%ebp)
mov    0xfffffffc(%ebp),%eax
movl   $0x8049440,(%eax)
leave
ret
nop
```

source code of **main**

```
void main() {
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

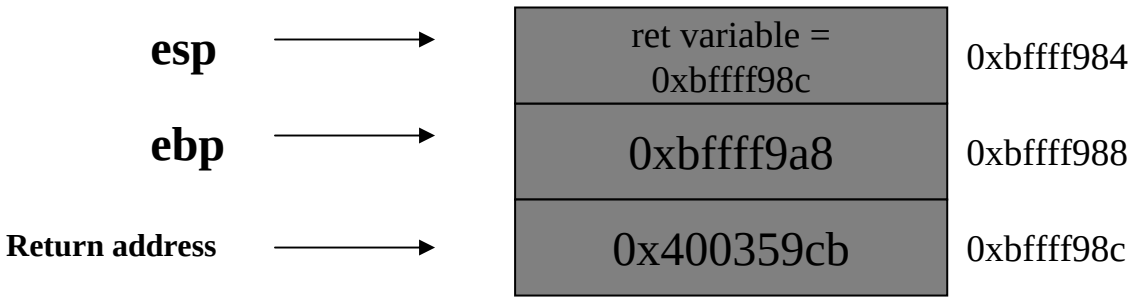


assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp
lea    0xfffffffc(%ebp),%eax   // load ax from ret value
lea    0x8(%eax),%edx          // offset two address and load to dx
mov    %edx,0xfffffffc(%ebp)   // update the ret variable in the stack
mov    0xfffffffc(%ebp),%eax
movl   $0x8049440,(%eax)
leave
ret
nop
```

source code of **main**

```
void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}
```

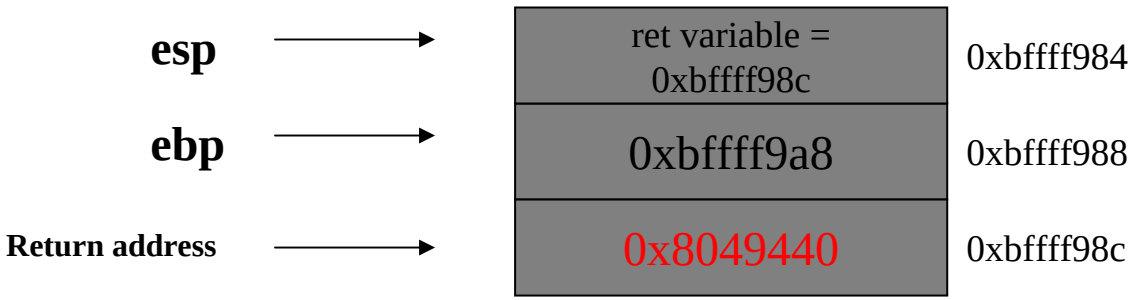


assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp
lea    0xfffffffc(%ebp),%eax   // load ax from ret value
lea    0x8(%eax),%edx          // offset two address and load to dx
mov    %edx,0xfffffffc(%ebp)   // update the ret variable in the stack
mov    0xfffffffc(%ebp),%eax   // move the ret value to ax
movl   $0x8049440,(%eax)       // move 0x8049440 value to
                               // stack position pointed by ax
leave
ret
nop
```

source code of **main**

```
void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}
```

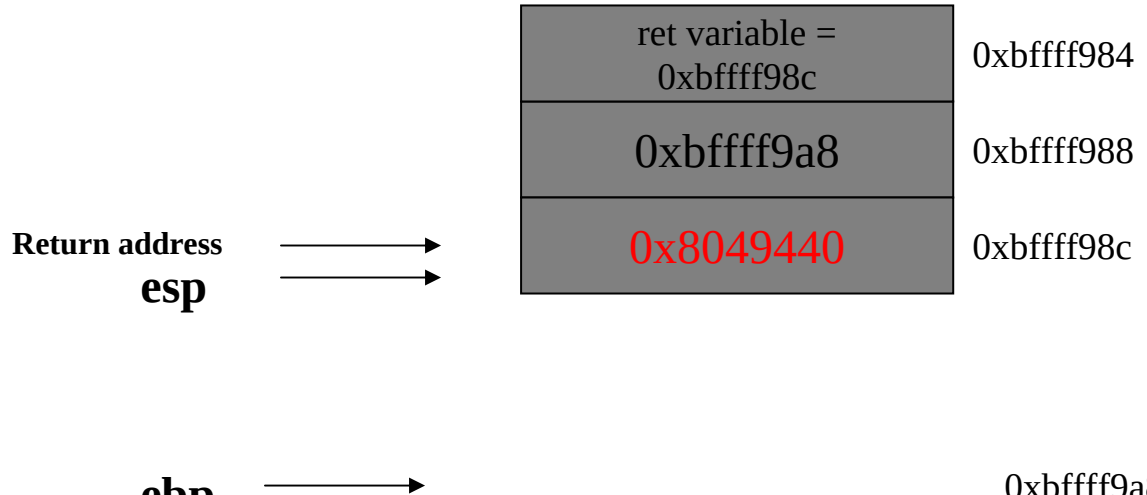


source code of **main**

```
void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}
```

assembler code of **main**

```
push   %ebp
mov    %esp,%ebp
sub    $0x4,%esp
lea    0xfffffffc(%ebp),%eax   // load ax from ret value
lea    0x8(%eax),%edx          // offset two address and load to dx
mov    %edx,0xfffffffc(%ebp)  // update the ret variable in the stack
mov    0xfffffffc(%ebp),%eax   // move the ret value to ax
movl   $0x8049440,(%eax)     // move 0x8049440 value to
                                // stack position pointed by ax
leave
ret
nop
```



Steps to Buffer Overflow Attack

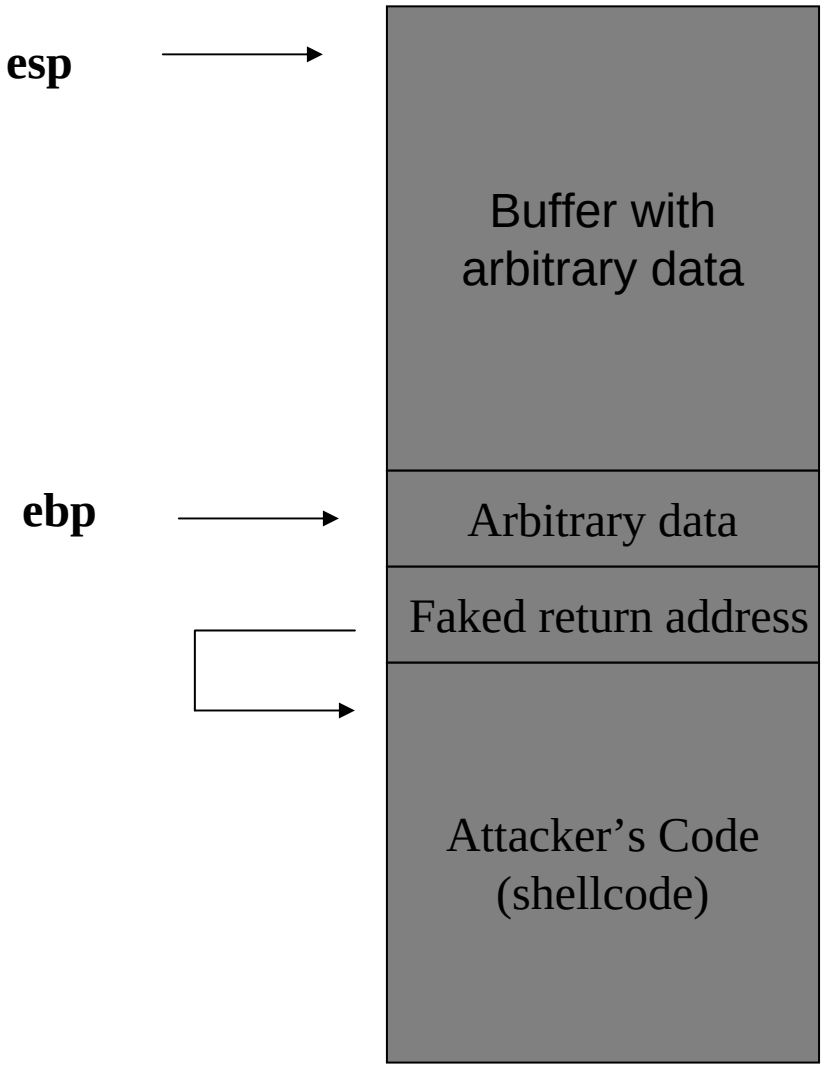
1. Discovering a code, which is vulnerable to a buffer overflow.
2. Determining the number of bytes to be long enough to overwrite the return address.
3. Calculating the address to point the alternate code.
4. Writing the code to be executed, usually the shell code.
5. Linking everything together and testing .

Difficulties in Buffer Overflow

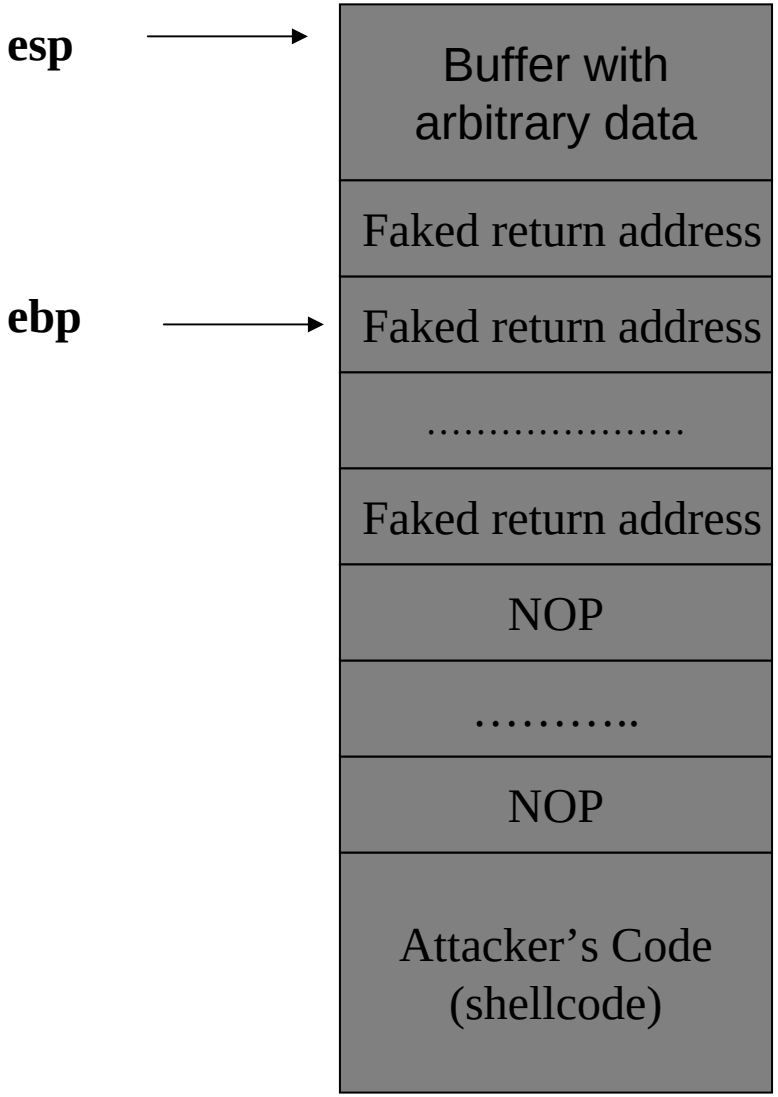
In order to make the buffer overflow succeed, we need to:

1. Calculate the right value to put into the fake return address
2. Calculate the location of the return address on the stack relatively to the overflowed buffer
3. Ensure the shellcode does not contain any zero

Stack after buffer overflow attack



Stuffing NOP on the Stack makes the buffer overflow attack easier



Buffer Overflow

Examples of local and remote root exploit through buffer overflow

- pwck local buffer overflow exploit
- QPOP 3.0beta AUTH remote root stack overflow

Prevention

- Non-executable stack
- suid wrappers
- Guard programs that check return addresses
- Bounds checking compilers, e.g. Libsafe
- StackGuard and Stack Shield in gcc
- WRITE SECURE CODE !!!
  - Always check the bounds of an array before writing it to a buffer.
  - Use functions that limit the number and/or format of input characters.
  - Avoid using dangerous C functions such as the following: scanf( ), strcpy( ), strcat( ), getwd( ), gets( ), strcmp( ), sprintf( ).