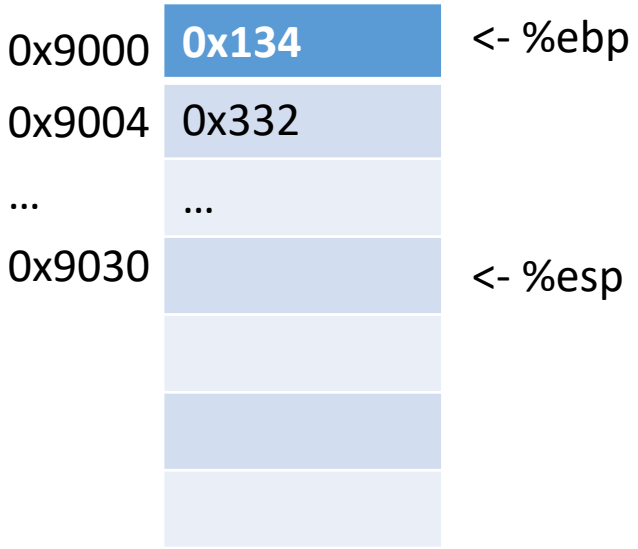
Lab 4: Buffer Overflow

Stack: Function call (1/2)

- Function A

```
{
        call Function B(parameter1)
}
```

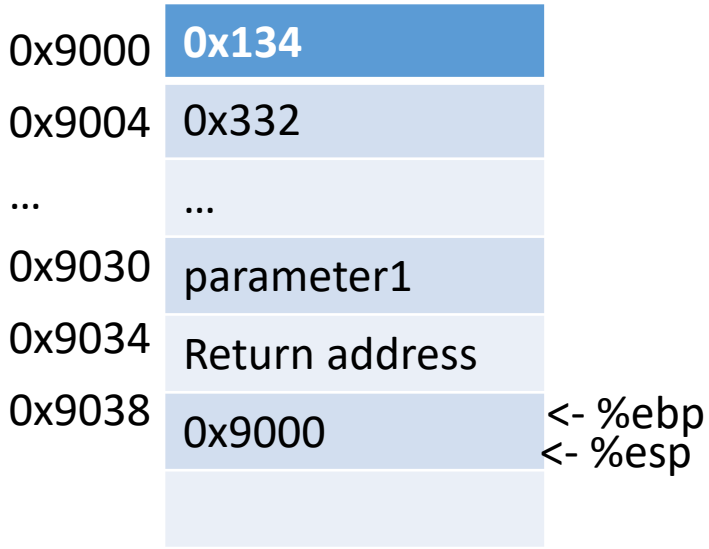


- Before calling the function B, %ebp points to base address of function A.

Stack: Function call (2/2)

- Function A

```
{
        call Function B(parameter1)
}
```



%esp	0x9038

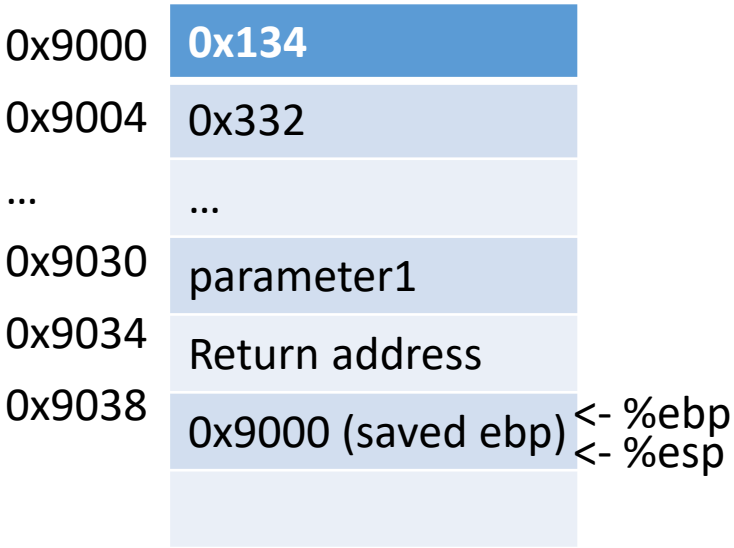
%ebp	0x9038

- Parameter1 is pushed onto stack
- When function B is called, the stack pointer for function A is stored.
- The stack pointer for function B is set

```
push %ebp
mov %esp, %ebp
```

Return address and Saved EBP Pointer

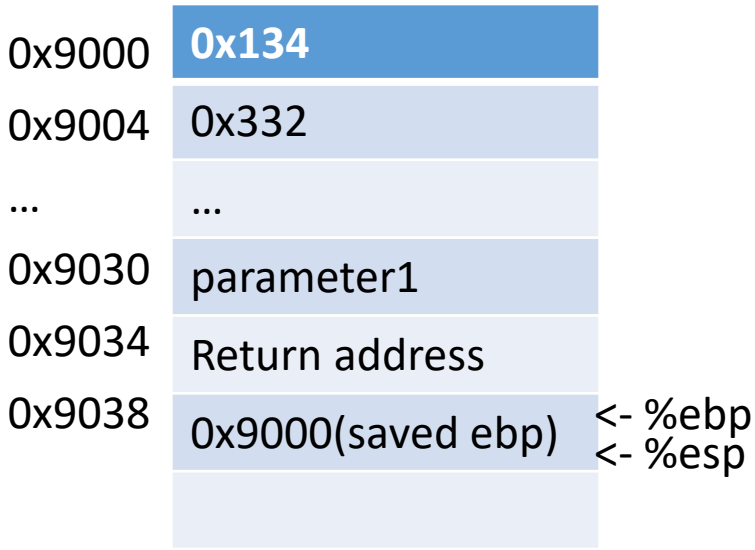
```
0x080491d4 <getbufn+0>: push   %ebp
0x080491d5 <getbufn+1>: mov    %esp,%ebp
0x080491d7 <getbufn+3>: sub    $0x208,%esp
0x080491dd <getbufn+9>: lea    -0x200(%ebp),%eax
0x080491e3 <getbufn+15>:        mov    %eax,(%esp)
0x080491e6 <getbufn+18>:        call   0x8048cal <Gets>
0x080491eb <getbufn+23>:        mov    $0x1,%eax
0x080491f0 <getbufn+28>:        leave
0x080491f1 <getbufn+29>:        ret
```



- When caller function calls callee function, the return address is pushed onto stack. (e.g. 0x080491eb)
- The value of %ebp of caller function is also pushed onto stack as saved EBP.

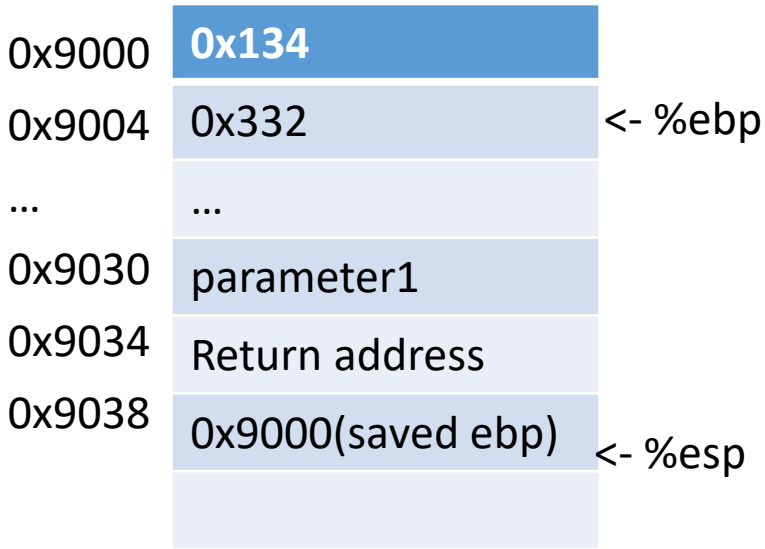
Stack: Function Return

- After finish to execute callee function, the %ebp is changed to saved %ebp.
- The address of instruction jump to return address (return to caller function



Stack: Function Return

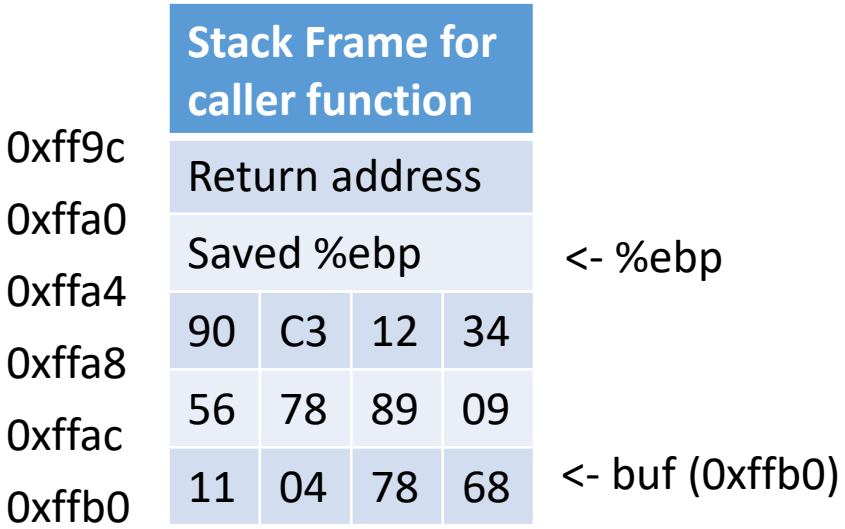
- After finish to execute callee function, the %ebp is changed to saved %ebp.
- The address of instruction jump to return address (return to caller function



Buffer Overflow (1/2)

- Buffer Overflow: while writing data to a buffer, overwrite the overruns the buffer's boundary and overwrite adjacent memory location
- "getbuf" function get input from stdin or file
- This function save the input in stack

```
void getbuf()
{
        char buf[12];
        gets(buf);
}
```



Buffer Overflow (2/2)

- If we insert input more than 12bytes, the input corrupts saved frame pointer and return address
- Jumps to address 0x00384855 when getbuf attempts to return
  - Invalid address, cause program to abort

	Stack Frame for caller function				
0xff9c	00	38	48	55	
0xffa0	32	11	23	22	<- %ebp
0xffa4	90	C3	12	34	
0xffa8	56	78	89	09	
0xffac	11	04	78	68	<- buf (0xffb0)
0xffb0					

GDB: Identifying Stack contents (1/2)

- x/<number>x <register>: identify stack work

```
(gdb) x/20x $esp
0x556833e8 <_reserved+1037288>: 0x556833f0      0x0063a685      0x61616161      0x61616161
0x556833f8 <_reserved+1037304>: 0x61616161      0x66647361      0x643b736b      0x6c6b666c
0x55683408 <_reserved+1037320>: 0x666b773b      0x736b3b6c      0x00647364      0x55683460
0x55683418 <_reserved+1037336>: 0x55683494      0x0063aa5d      0x00661cf9      0x08048564
0x55683428 <_reserved+1037352>: 0xf7fd92e8      0x00000002      0xf7fd9010      0x006501a4
```

- Info register (i r): identify the value of registers

```
(gdb) i r
eax            0x1      1
ecx            0xa      10
edx            0x46     70
ebx            0x55686028       1432903720
esp            0x556835f4       0x556835f4
ebp            0x55683610       0x55683610
esi            0x64cca0 6605984
edi            0x0      0
eip            0x80491f1        0x80491f1 <getbufn+29>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```

GDB: Identifying Stack content2 (2/2)

- Info frame (info f): This command prints a verbose description of the selected stack frame, including:
  - the address of the frame
  - the address of the next frame down (called by this frame)
  - the address of the next frame up (caller of this frame)
  - the language in which the source code corresponding to this frame is written
  - the address of the frame's arguments
  - the address of the frame's local variables
  - the program counter saved in it (the address of execution in the caller frame)
  - which registers were saved in the frame

Practice: Find local buf address

- To exploit buffer overflow, we have to know the start address of local buf (*i.e where the input is stored in stack from getbuf function*)
- This address gives the information of how many input we have to insert to make buffer overflow

Practice: Return to Smoke Function

- Original code has to return to test function.
- By exploiting buffer overflow, change the return address to jump to smoke function

```
1  void test()
2  {
3      int val;
4      /* Put canary on stack to detect possible corruption */
5      volatile int local = uniqueval();
6
7      val = getbuf();
8
9      /* Check for corrupted stack */
10     if (local != uniqueval()) {
11                 printf("Sabotaged!: the stack has been corrupted\n");
12     }
```

```
1  /* Buffer size for getbuf */
2  #define NORMAL_BUFFER_SIZE 32
3
4  int getbuf()
5  {
6      char buf[NORMAL_BUFFER_SIZE];
7      Gets(buf);
8      return 1;
9  }
```

```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

Practice: Return to Smoke Function

- Original code has to return to test function.
- By exploiting buffer overflow, change the return address to jump to smoke function

```
1  void test()
2  {
3      int val;
4      /* Put canary on stack to detect possible corruption */
5      volatile int local = uniqueval();
6
7      val = getbuf();
8
9      /* Check for corrupted stack */
10     if (local != uniqueval()) {
11                 printf("Sabotaged!: the stack has been corrupted\n");
12     }
```

```
1  /* Buffer size for getbuf */
2  #define NORMAL_BUFFER_SIZE 32
3
4  int getbuf()
5  {
6      char buf[NORMAL_BUFFER_SIZE];
7      Gets(buf);
8      return 1;
9  }
```

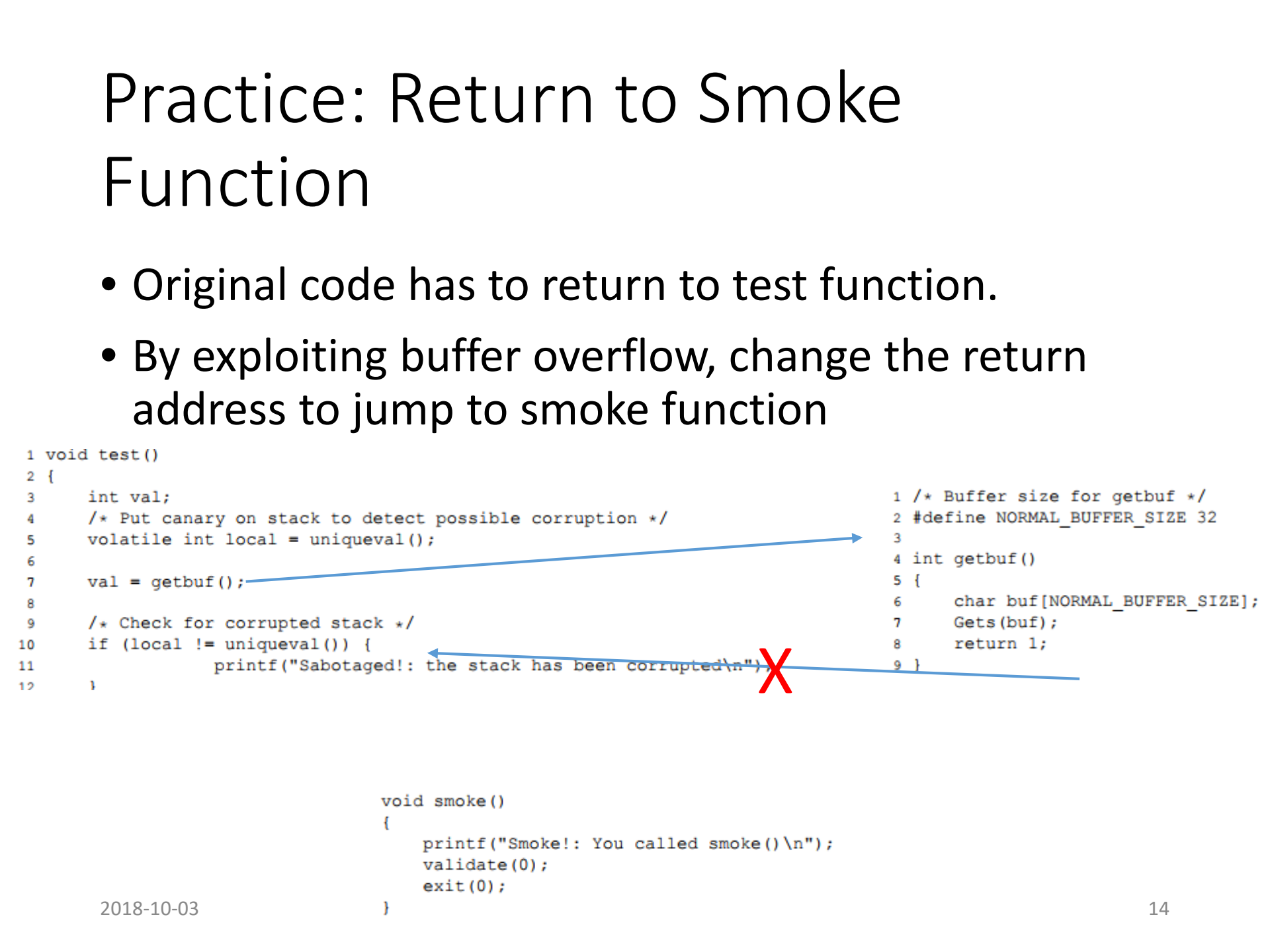
```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

Practice: Return to Smoke Function

- Original code has to return to test function.
- By exploiting buffer overflow, change the return address to jump to smoke function

```
1  void test()
2  {
3      int val;
4      /* Put canary on stack to detect possible corruption */
5      volatile int local = uniqueval();
6
7      val = getbuf();
8
9      /* Check for corrupted stack */
10     if (local != uniqueval()) {
11                 printf("Sabotaged!: the stack has been corrupted\n");
12     }
```

```
1  /* Buffer size for getbuf */
2  #define NORMAL_BUFFER_SIZE 32
3
4  int getbuf()
5  {
6      char buf[NORMAL_BUFFER_SIZE];
7      Gets(buf);
8      return 1;
9  }
```



```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

Practice: Return to Smoke Function

- Original code has to return to test function.
- By exploiting buffer overflow, change the return address to jump to smoke function

```
1  void test()
2  {
3      int val;
4      /* Put canary on stack to detect possible corruption */
5      volatile int local = uniqueval();
6
7      val = getbuf();
8
9      /* Check for corrupted stack */
10     if (local != uniqueval()) {
11                 printf("Sabotaged!: the stack has been corrupted\n");
12     }
```

```
1  /* Buffer size for getbuf */
2  #define NORMAL_BUFFER_SIZE 32
3
4  int getbuf()
5  {
6      char buf[NORMAL_BUFFER_SIZE];
7      Gets(buf);
8      return 1;
9  }
```

```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

Buffer Overflow Lab

- Make buffer overflow with following condition
- There are 4 levels
- Using the practices to find local buffer address
- Reference additional pdf file
- Due date: 10/16 11:59