Honeypot

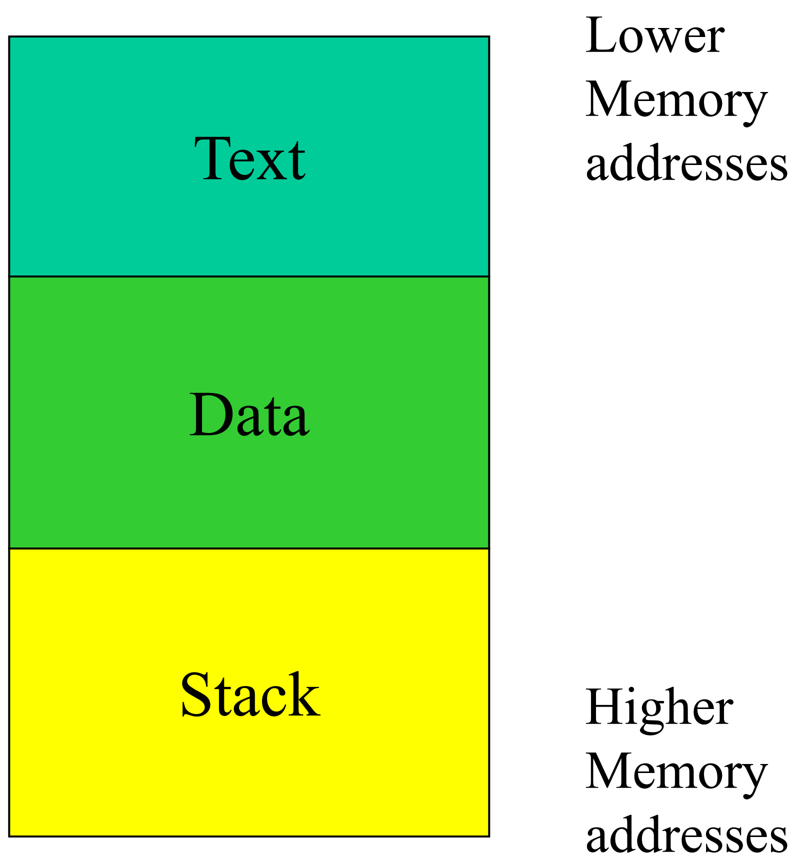
Introduction

- When a program is executed, the system maintains an **execution stack** to store the information about the active functions in the program.
- One important information stored in the execution stack is the address of the next instruction to be executed when the function terminates.

- In many C/C++ implementations, it is possible to corrupt this **execution stack** by writing past the end of an array.
- Known as **smash the stack**.
  - When the function terminates, the control flow will jump to a random address in the memory.
  - A hacker can control the program flow by using carefully crafted set of data to write past the end of the array.
  - We will demonstrate that the hacker can obtain the **root privilege** in Linux by using this technique.

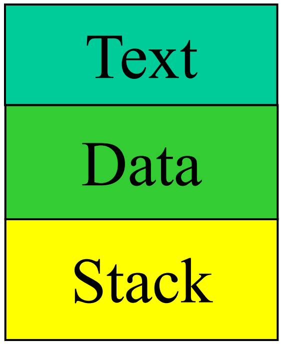
- Buffer overflow has been discovered for over 20 years and many efforts have been made to fix such vulnerabilities
- However, many modern systems nowadays may still be vulnerable
  - Most operating systems nowadays are fully or at least partially written in C/C++
  - Many existing software are written in C/C++
    - Microsoft Office
    - Various computer games
      - Many game engines are written in C++, e.g., Unity, Unreal Engine, CryENGINE
- When we develop/maintain a software/an operating system, we still need to pay attention to this vulnerability

Process Memory Organization



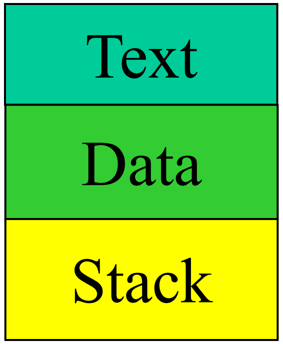
Process Memory Regions

- 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
  - Last in, first out (LIFO)

- Stack is used to
  - Dynamically allocate the local variables used in functions.
  - Pass parameters to the functions.
  - Return values from the functions.
- Stack Pointer (SP) points to the top of the stack.
- The bottom of the stack is at a fixed address.
- The stack consists of Logical Stack Frames that are pushed when calling a function and popped when returning.
- Frame Pointer (FP) points to a fixed location within a frame.



- Let's see an example: example1.c

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

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

- Get the assembly code by

  gcc –S –o example1.s example1.c

example.s

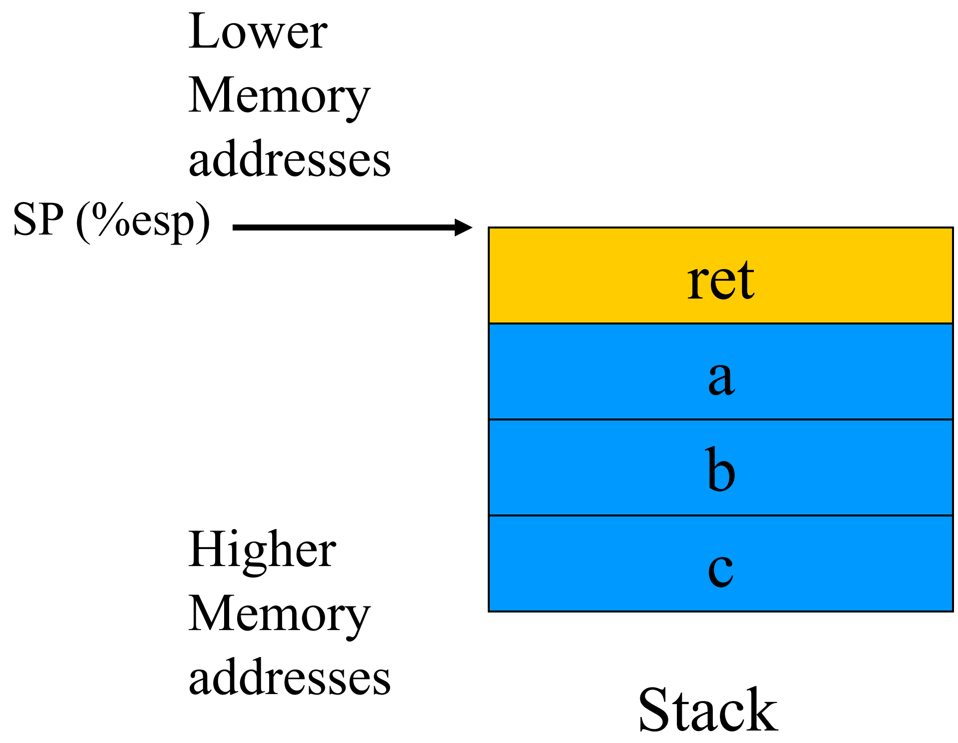
- Calling function is translated into

  pushl $3

  pushl $2

  pushl $1

  call function



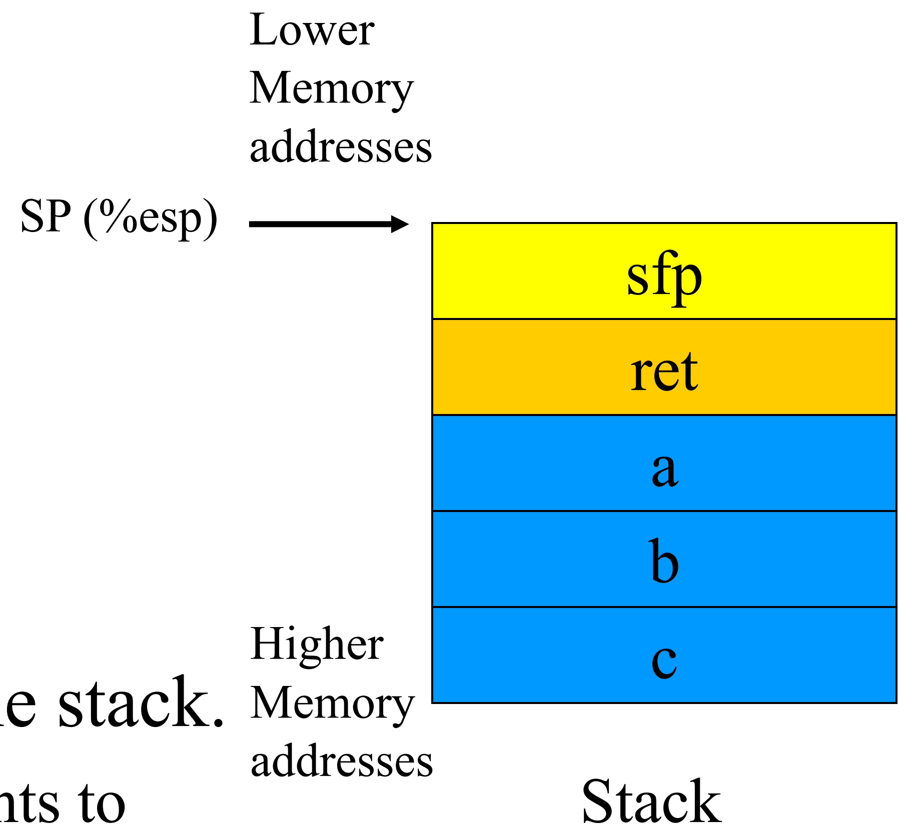
  - Its pushes the 3 arguments backwards into the stack.
  - The instruction ‘call’ will push the Instruction Pointer (IP) onto the stack.
    - We call the saved IP in stack as the **Return Address** (RET), which will be referred after executing the function.

- Procedure prolog

  pushl %ebp

  movl %esp, %ebp

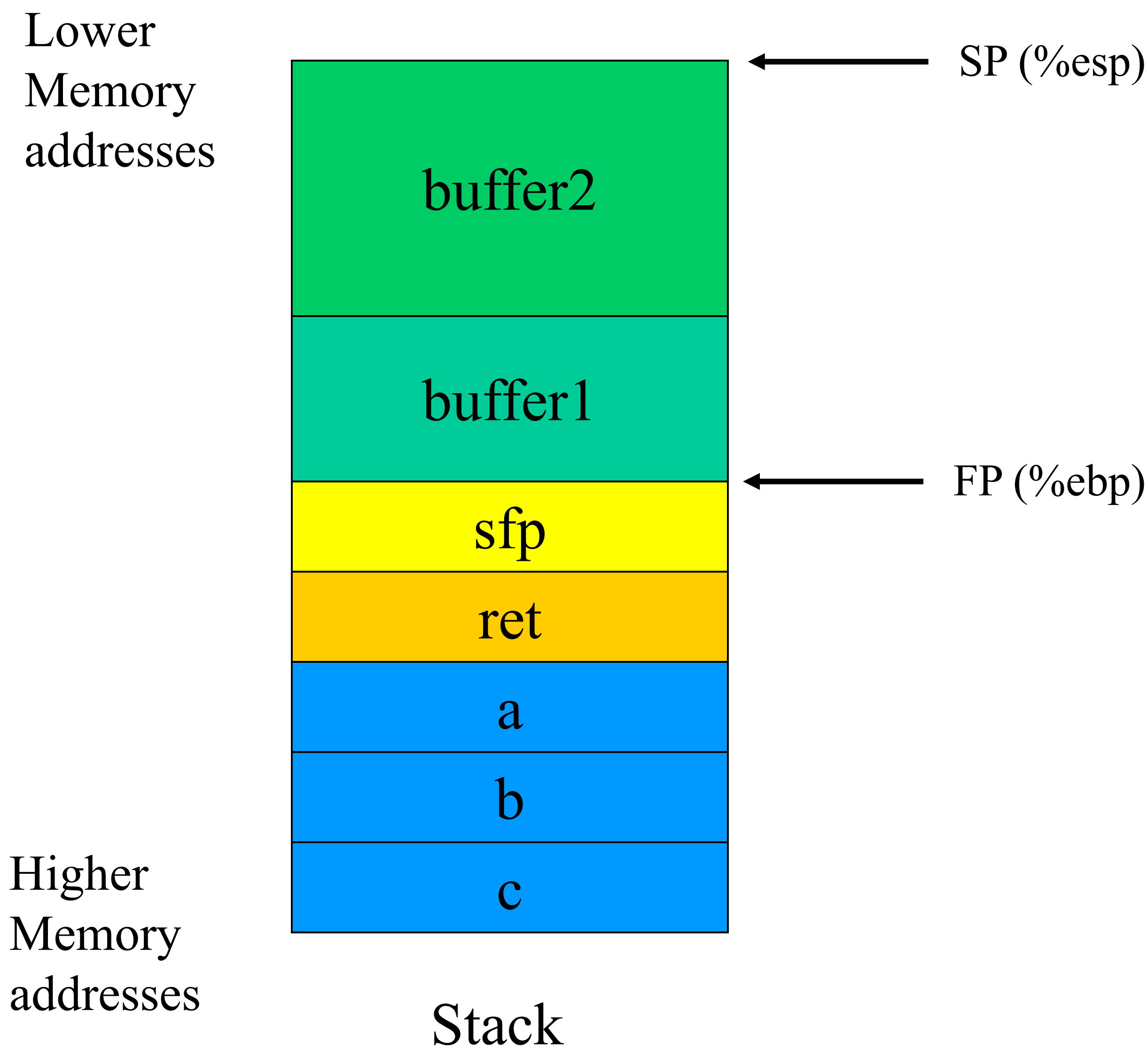
  subl $20, %esp



- Pushes the old FP onto the stack.
  - %ebp store FP, which points to the current stack frame
  - We call the saved FP in stack as SFP.

- Copies the current SP onto %ebp, make it the new FP.
  - %esp stores SP, which points to the top of the stack.

- Allocates space for the local variables by subtracting their size from SP.
  - Memory is addressed in multiples of the word size.
  - In our case, the word size is 4 bytes.
  - 5 byte buffer take 8 bytes (2 words).
  - 10 byte buffer take 12 bytes (3 words).
  - SP is subtracted by 20.
  - Note that the address calculation varies in different systems, compilers, and compiler settings.

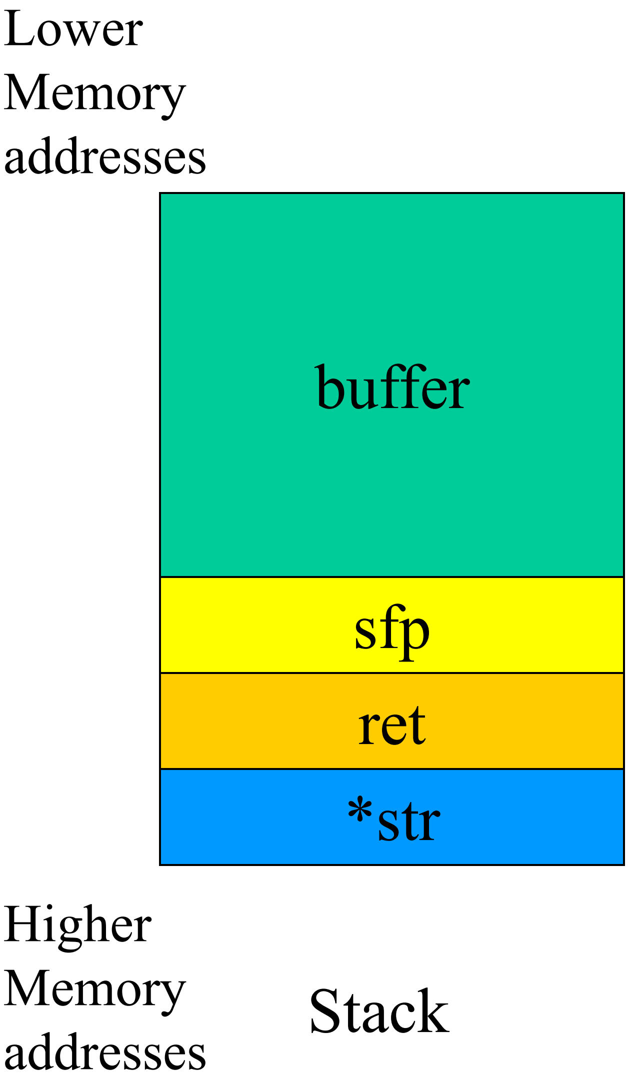

Lower
Memory
addresses
buffer2
buffer1
sfp
ret
a
b
c
Higher
Memory
addresses
SP (%esp)
FP (%ebp)
Stack

Buffer Overflows

- Result of stuffing more data into a buffer than it can handle.
- See the following example:

example2.c

- It copies a string without bound checking by using strcpy() instead of strncpy().
  - Copy the contents of *str into buffer[] until a null character is found.



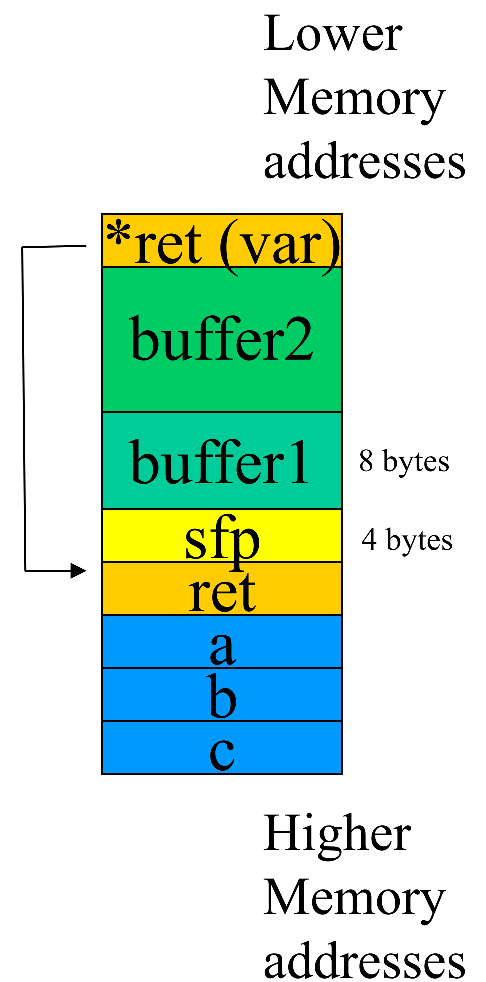
- buffer[] (16 bytes) is much smaller than *str (256 bytes).
- All 239 bytes after buffer in the stack are being overwritten with character 'A' (0x41)
  - Include SFP, RET and even *str.
  - The return address becomes 0x41414141.
  - When the function returns, there will be a segmentation fault as that address does not belong to this process (most probably)

What is the trick?

- Buffer overflow allows us to change the return address of a function.

example3.c

  - ret = buffer1 + 12
    the pointer *ret is now pointing to the return address of the function (ret in the diagram).
  - (*ret) += 8
    the return address of the function is increased by 8.
  - Skip the assignment statement to the printf call.
  - The program prints 0 instead of 1.
- Due to different computer configurations, the following is used for our current machines.
  - ret = buffer1 + 28
  - (*ret) += 10



- By using a similar technique, a hacker can change the return address, so that the control flow will pass to the hacker's desired code.
- The code will be run under the username of the owner of the program.
- Usually the hacker wants to have a shell to issue some commands later.
- The machine code to generate a shell is called **shellcode**.
- Read the article by Aleph One, to see how the shell code is obtained.
- Here is an example to test the shellcode.

testsc2.c

- How can we make **ret** (in the diagram) point to the shellcode?
- Note that:

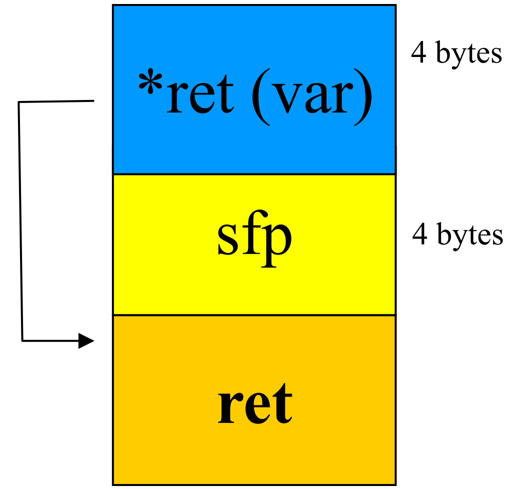
```
ret = (int *)&ret + 2;
```

is actually adding 8 bytes to &ret.

- So that the *ret (var) pointer will point to **ret** in the diagram
- After that, the code

```
(*ret) = (int)shellcode;
```

will write the address of the shellcode to **ret** in the stack



Writing an Exploit

- Suppose we want to "overflow" this program, which is owned by root.

vulnerable.c

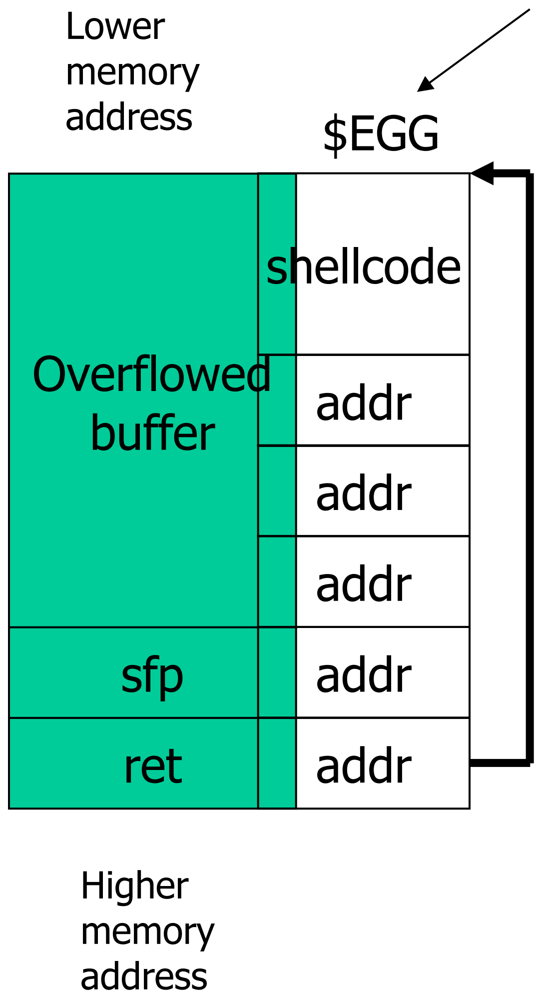
- The hacker creates an exploit program to put the overflow string in an **environment variable**.
  - The environment variables are stored near the stack when the vulnerable program is executed.

- The exploit program takes a buffer size and an offset as parameters
  - Guess the position of the buffer we want to overflow.

Exploit2

Generated by exploit2.c

Stack diagram for vulnerable.c



addr tries to point to the beginning of the shellcode (the chance is very low though)

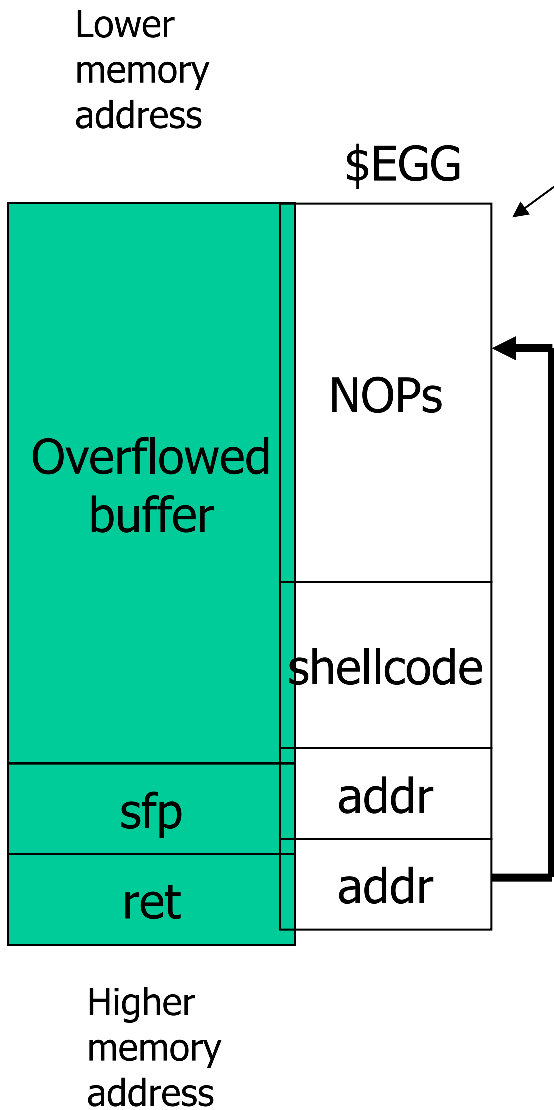
In exploit2.c, the meaning of setuid(0) before the shellcode will be explained later

- It is very difficult for exploit2 to success
- The problem is that we need to guess exactly where the address of our shellcode will start.
- To increase our chance:
  - Pad the front of our overflow buffer with NOP instructions (i.e., no operations).
  - Fill half of the buffer with NOP.
  - Place the shellcode in the middle.
  - Then fill the addresses.
- If the return address points anywhere in the string of NOPs, the NOP instructions will be executed, until the shellcode is reached.

Exploit3

Stack diagram for vulnerable.c

Generated by exploit3.c

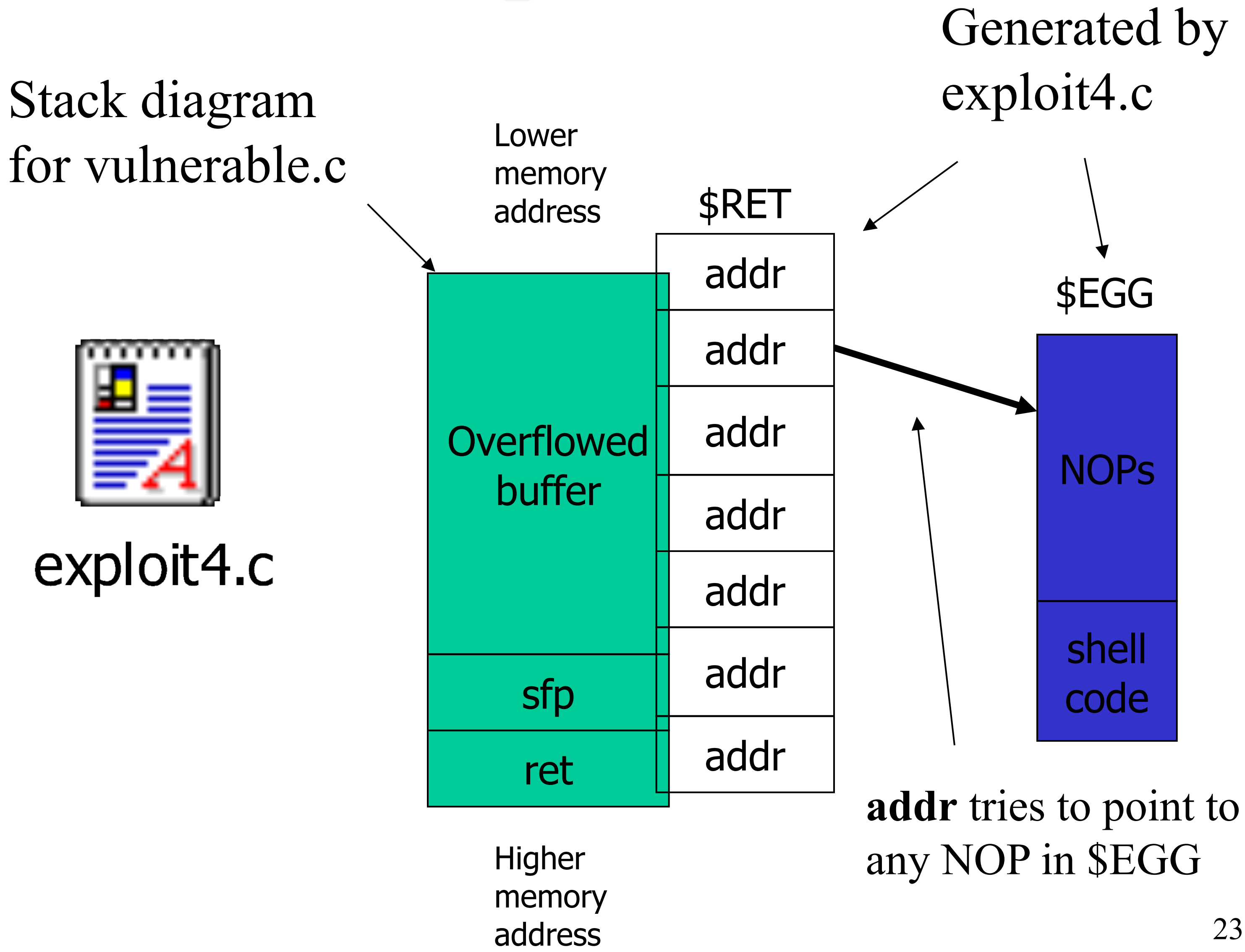


addr tries to point to any NOP (much easier compare with exploit2)

- What if the buffer is too small for the shellcode to fit into it?
  - The size of our shellcode is around 50 bytes
  - Suppose the size of the buffer in vulnerable.c is now 30 instead of 512
  - Even if we do not pad NOP, part of the shellcode is already overwriting ret

- Solution
  - Put the shellcode in another place.
  - Use two environment variables:
    - One contains the shellcode.
    - Another contains the addresses trying to point to the shellcode

Exploit4


Stack diagram
for vulnerable.c
Generated by
exploit4.c
exploit4.c
Lower
memory
address
$RET
addr
addr
addr
addr
addr
addr
addr
$EGG
Overflowed
buffer
sfp
ret
NOPs
shell
code
addr tries to point to
any NOP in $EGG
Higher
memory
address

- If the vulnerable program is owned by root with the setuid permission set, we can access the root account.
- Trial

```
[mhwong]$ ./exploit4 768
using address: 0xbffff9d8
[mhwong]$ ./vulnerable $RET
bash# whoami
root
bash#
```

Hacker: “Ha! Ha! Ha! …I am the root now!!”

Reference

- Aleph One, “Smashing The Stack for Fun and Profit”, Phrack 49 Volume 7, Issue 49, File 14 of 16.