Smashing the stack
-
continued

Last week

- Using buffer overruns or format string attacks to read or write things on the stack we're not supposed to
  - esp. the control data on the stack:

    frame pointers and return addresses

- A classic buffer overflow to
  1. first inserting malicious shell code into some buffer on the stack
  2. then overwriting return address on the stack to then jump to that code

Today:

- some variations when messing with frame pointers and return addresses
- some defenses

warning: potential exam questions coming up

example vulnerable code

```
m(){
 int x = 4;
 f(); // return_to_m
 printf (“x is %i”, x);}

f(){
 int y = 7;
 g(); // return_to_f
 printf (“y+10 is %i”, y+10);}

g(){
 char buf[80];
 gets(buf);
 printf(buf);
 gets(buf);}
```

example vulnerable code

```
m(){
 int x = 4;
 f(); // return_to_m
 printf (“x is %i”, x);   }

f(){
 int y = 7;
 g(); // return_to_f
 printf (“y+10 is %i”, y+10); }

g(){
 char buf[80];
 gets(buf);
 printf(buf);
 gets(buf); }
```

An attacker could

1. first inspect the stack using a malicious format string
   (entered in first **`gets`** and printed with **`printf`**)
2. then overflow **`buf`** to corrupt the stack
   (with the second **`gets`**)

potential overflow of **`buf`**

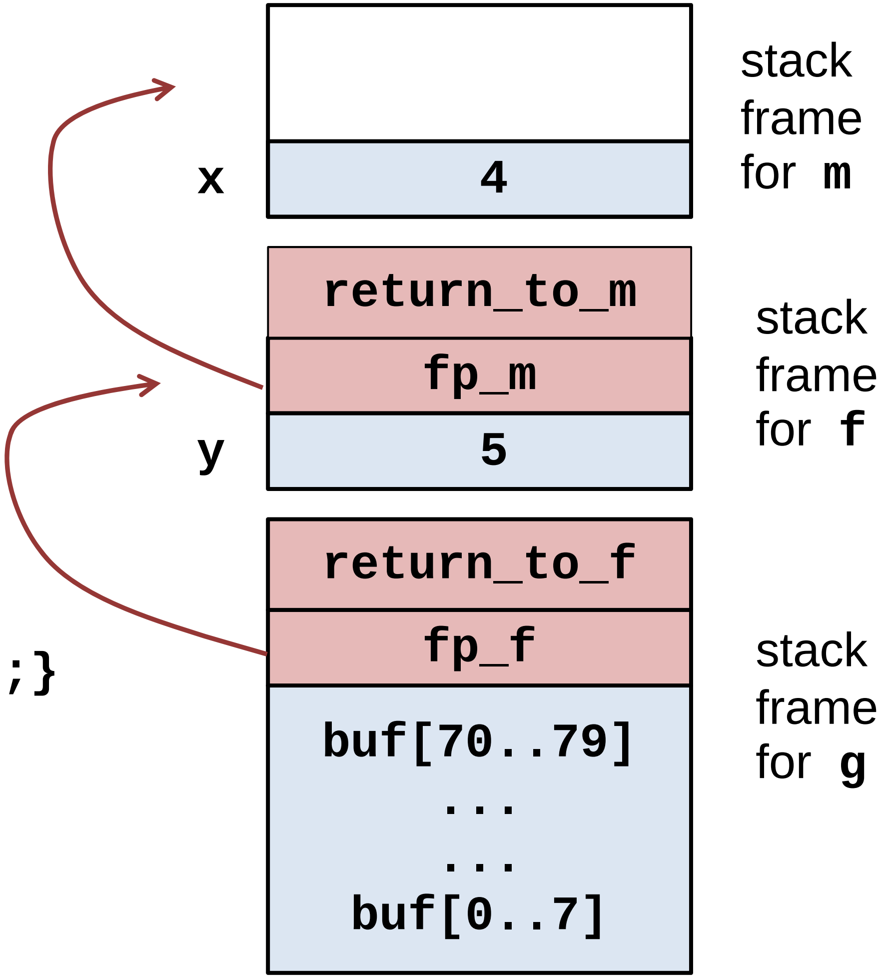
potential format string attack

example vulnerable code

```
m(){
 int x = 4;
 f(); // return_to_m
 printf (“x is %i”, x);}

f(){
 int y = 7;
 g(); // return_to_f
 printf (“y+10 is %i”, y+10);}

g(){
 char buf[80];
 gets(buf);
 printf(buf);
 gets(buf);}
```



x 4

stack frame for **m**

return_to_m

fp_m

y 5

stack frame for **f**

return_to_f

fp_f

buf[70..79]

...

...

buf[0..7]

stack frame for **g**

Normal execution

- After completing **`g`**
  execution continues with **`f`** from program point **`return_to_f`**

  This will print 17.

- After completing **`f`**
  execution continues with **`main`** from program point **`return_to_m`**

  This will print 4.

If we start smashing the stack different things can happen

Attack scenario 1

in **`g()`** we overflow **`buf`** to overwrite values of **`x`** or **`y.`**

- After completing **`g`**
  execution continues with **`f`** from program point **`return_to_f`**

  This will print whatever value we gave to **`y`** +10.

- After completing **`f`**
  execution continues with **`m`** from program point **`return_to_m`**

  This will print whatever value we gave to **`x.`**

Of course, it is easier to overwrite local variables
in the current frame than variables in lower frames

Attack scenario 2

In **`g()`** we overflow **`buf`** to overwrite return address **`return_to_f`** with **`return_to_m`**

- After completing **`g`**
  execution continues with **`m`** instead of **`f`**
  but with **`f`**'s stack frame.

  This will print 7.

- After completing **`m`**
  execution continues with **`m`**.

  This will print 4.

Attack scenario 3

In **`g()`** we overflow **`buf`** to overwrite frame pointer **`fp_f`** with **`fp_m`**

- After completing **`g`**
  execution continues with **`f`**
  but with **`m`**'s stack frame

  This will print 14.

- After completing **`f`**
  execution continues with whatever code called **`m.`**

  So we never finish the function call **`m`** , the remaining part of the code (after the call to **`f`**) will never be executed.

Attack scenario 4

In **`g()`** we overflow **`buf`** to overwrite frame pointer **`fp_f`** with **`fp_g`**

- After completing **`g`**
  execution continues with **`f`**
  but with **`g`**'s stack frame.

  This will print (some bytes of **`buf`** +10).

- After completing **`f`**, execution might continue with **`f`**,
  again with **`g`**'s stack frame, repeating this for ever.

  This depends on whether the compiled code looks up values from the top of **`g`**'s stack frame, or the bottom of **`g`**'s stack frame. In the latter case the code will jump to some code depending on the contents of **`buf`**.

Attack scenario 5

In **`g()`** we overflow **`buf`** to overwrite frame pointer **`fp_f`** with some pointer into **`buf`**

- After completing **`g`**
  execution continues with **`f`**
  but with part of **`buf`** as stack frame.

  This will print (some part of **`buf)+10.`**

- After completing **`f`**
  not clear what will happen...

Attack scenario 6

In **`g()`** we overflow **`buf`** to overwrite the return address **`return_to_f`** to point in some code somewhere, and the frame pointer to point inside **`buf.`**

- After completing **`g`**
  execution continues executing that code
  using part of **`buf`** as stack frame.

  This can do all sorts of things!
  If we have enough code to choose from, this can do anything we want.

Often a return address in some library routine in libc is used,
in what is then called a return-to-libc attack.

Attack scenario 7

In **`g()`** we overflow **`buf`** to overwrite the return address to point inside **`buf`**

- After completing **`g`** execution continues with whatever code (aks shell code) was written in **`buf`** ,

  using **`f`**'s stack frame.

  This can do anything we want.

This is the classic buffer overflow attack

- You could also overwrite **`sp_f`** and supply the attack code with a fake stack frame, but typically the shell code won't need a stack frame
- This attack requires that the computer (OS+ hardware) can be tricked into executing data allocated on the stack. Many systems will no longer execute data (code) on the stack or on the heap.

Memory segments

Normally (always?) the program counter should point somewhere in the code segment

The attack scenarios discussed in these slides only involved overflowing buffers on the stack.

Buffers allocated on the heap or global buffers can also be overflowed to change program behaviour, but to mess with return addresses or frame pointers we need to overflow on the stack