Attacking the stack

Thanks to SysSec and Secure Systems Labs
at Vienna University of Technology for some of these slides

Attacking the stack

We have seen how the stack works.

Now: let’s see how we can abuse this.

We have already seen how code (incl. malware) can *deliberately* do ‘strange things’,

- accessing raw memory representations
- manipulate memory *anywhere* on the heap and stack

Now: let’s see how benign, but buggy code can be *manipulated* into doing strange things by malicious input

We’ll show two techniques for this

1. buffer overflows
2. format strings attacks

Attacking the stack

Goals for an attacker

1. leaking data - eg HeartBleed, or just last week Cloudbleed
2. corrupting data
3. corrupting program execution

   This can be

   3a) crashing

   3b) doing something more interesting

In *CIA terminology*, such attacks result in breaking

1. *confidentiality of data*
2. *integrity of data*
3. *integrity of program in execution (ie the “process”)*
4. *availability of data or the process*

   (if data is destroyed or program crashes)

Format string attacks

Format strings attacks

- Format string attacks were only discovered (invented?) in 2000, after people had been programming in C for over 25 years!
- These attacks allow an attacker to read or to corrupt the stack

- Not such a big problem as buffer overflows, as potential for format string attacks is *easy to spot and remove*
  - format attacks *should* be history by now...

- Still, a great example of how some harmless looking code can turn out to be vulnerable

Leaking data

```
int main( int argc,  char** argv)
    int pincode = 1234;
    printf(argv[1]);
}
```

This program echoes the first program argument.

Aside on `main(int argc, char** argv)`

`argc` is the numbers of arguments, **`argv`** are the argument values.

`argv` has type is a **`char**,`** so it is a pointer to a pointer to a **`char`**

`*argv` has type **`char*`** (ie a string)

`**argv` has type **`char`**

and using pointer arithmetic

`argv[i]` has type **`char*`**, ie a string

`argv[i][j]` has type **`char`**,

So effectively **`argv`** is an array of strings, or a 2-dimensional array of **`char`**'s

Note that

- when you call an executable from the command line, then **`argv[0]`** is the name of the executable, and **`argv[1]`** is the first real argument
- **`char** argv`** can also be written as **`char **argv`**

format strings for `printf`, using the `%` character

```
printf( "j is %i.\n" , j);
     // %i to print integer value
printf( "j is %x in hex.\n" , j);
     // %x to print 4-byte hexadecimal value
```

`"j is %i "` is called a format string

Other printing functions, eg **`snprintf`**, also accept format strings.

Any guess what

```
    printf("j is %x in hex");
```

does?

It will print the top 4 bytes of the stack

Leaking data with format string attack

```
int main( int argc,  char** argv)
    int pincode = 1234;
    printf(argv[1]);
}
```

This program may leak information from the stack when given *malicious input*, namely an argument that contains *special control characters*, which are *interpreted* by **`printf`**

Eg supplying **`%x%x%x`** as input will dump top 12 bytes of the stack

Leaking data from memory – using strings

```
printf( ”j is %s.\n” , str);
     // %s to print a string, ie a char*
```

Any guess what

```
    printf(”j is %s in hex”);  // %s instead of %i
```

does?

It will interpret the top of the stack as a pointer (an address) and will print the string allocated in memory at that address

Of course, there might not be a string allocated at that address, and `printf` simply prints whatever is in memory up to next null terminator

Corrupting data with format string attack

```
int j;
char* msg; ...
printf( ”how long is %s anyway %n” , msg, &j);
```

`%n` causes the number of characters printed to be written to j,

here it will write `20+length(msg)`

Any guess what

```
printf(”how long is this %n”);
```

does?

It interprets the top of the stack as an address, and writes a value there

Example malicious format strings

Interesting inputs for the string **`str`** to attack **`printf(str)`**

- **`%x%x%x%x%x%x%x%x`**

  will print bytes from the top of the stack

- **`%p%p%p%p%p%p%p%p`**

  will print these bytes as pointer values

- **`%s`**

  will interpret the top bytes of the stack as an address X, and then prints the string starting at that address A in memory, ie. it dumps all memory from A up to the next null terminator

- **`%n`**

  will interpret the top bytes of the stack as an address X, and then *writes* the number of characters output so far to that address

Example *really* malicious format strings

An attacker can try to control which address X is used for reading from memory using **`%s`** or for writing to memory using **`%n`** with specially crafted format strings of the form

- **`\xEF\xCD\xCD\xAB %x %x   ... %x %s`**
  With the right number of **`%x`** characters, this will print the string located at memory address **`ABCDCDEF`**

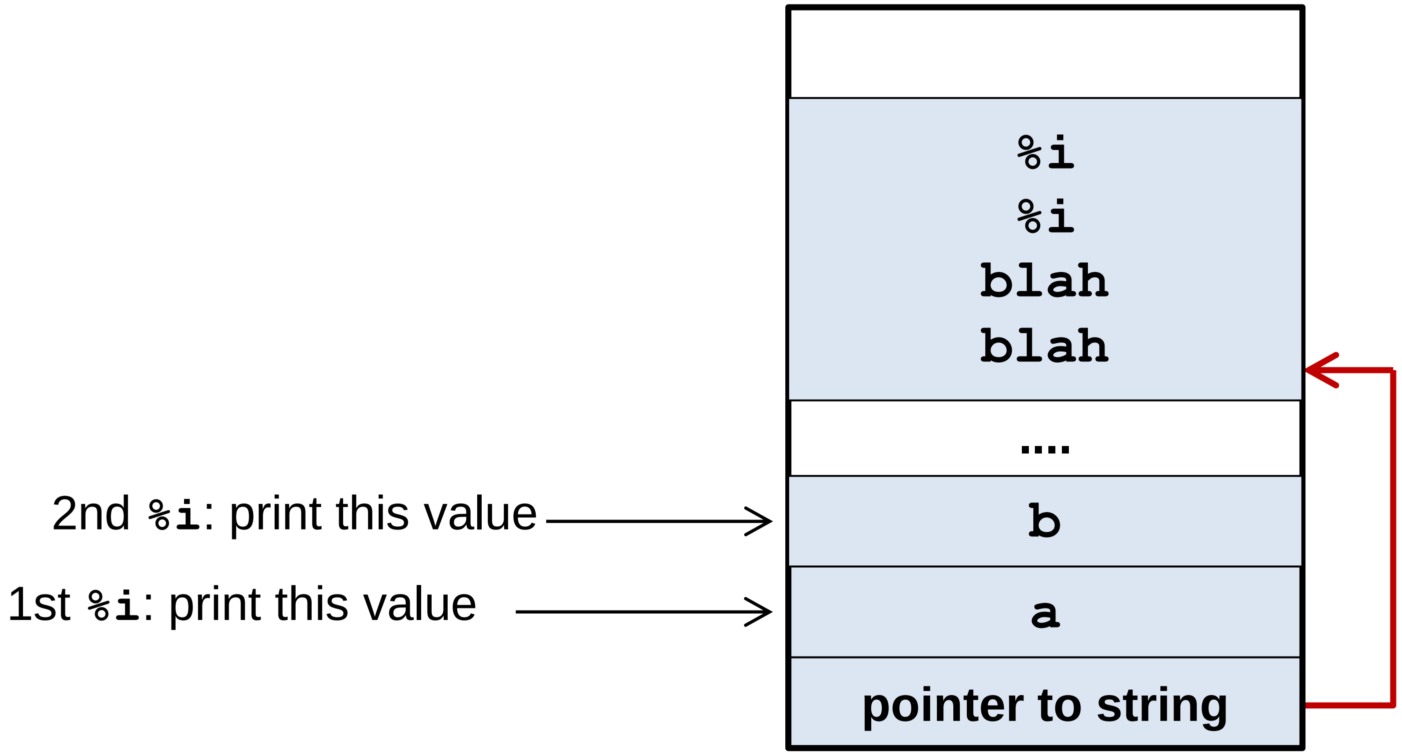
- **`\xEF\xCD\xCD\xAB %x %x   ... %x %n`**
  With the right number of **`%x`** characters, this will write the number of characters printed so far to memory address **`ABCDCDEF`**

The tricky things are inserting the right number of **`%x`**, and choosing an interesting address

stack layout for printf

`printf("blah blah %i %i",  a, b)`

Recall: string is written upwards



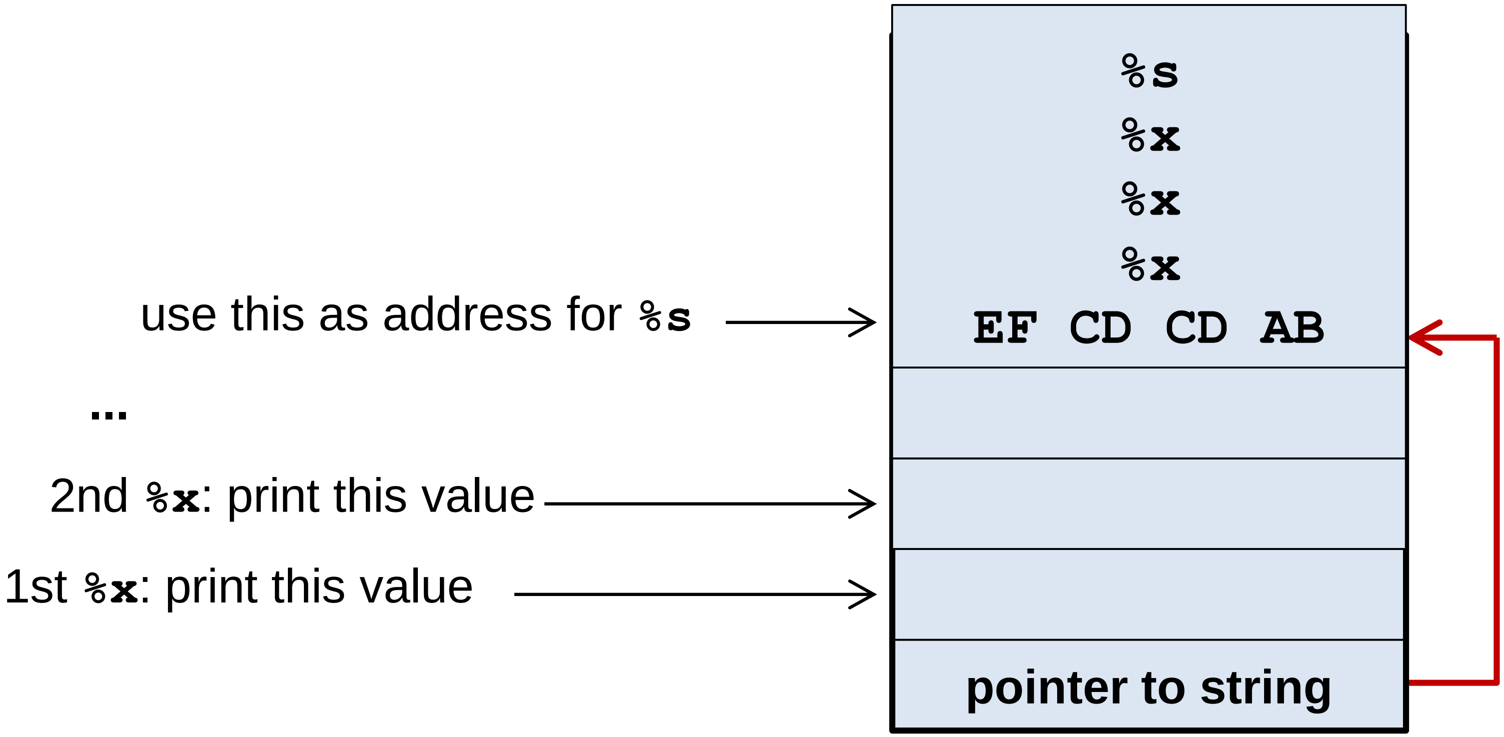
2nd **`%i`**: print this value

1st **`%i`**: print this value

stack layout for really malicious strings

`printf("\xEF\xCD\xCD\xAB %x %x ... %x %s");`

With the right number of **`%x`**'s, this will print the string located at address **`ABCDCDEF`**



Format strings attacks are easy to get rid of!

- Potentially vulnerable code is easy to spot

```
printf(str);                    // unsafe
printf("Some string literal");
printf("Some integer %i",n);
printf("%s",str);              // safe equivalent
```

- *Only the first statement is potentially vulnerable*
  - namely, if `string` is or contains user-supplied input
    aka `string` is untrusted or tainted

- First and last statement have same effect, so unsafe first statement can be replaced by the safe last statement, getting rid of any format string vulnerabilities

- This has to be done for *all* functions in the `..print..` family

buffer overflows

Buffer overflows

It is easy to make mistakes using arrays, pointers and strings, and accidentally read or write memory you shouldn't

- going outside array bounds
- copying a string into buffer where it does not fit
- having a string without a NULL terminator
  - string operations, such as **`printf`** and **`strcopy`**, assume there is a NULL, and will go off the rails if there is none
- having a pointer pointing to the wrong place, eg
  - a stale pointer that points to memory that has been freed
  - a mistake in your pointer arithmetic
  - ...

Typical string problem – incorrect buffer length

```
void vulnerable(char *s){
 char buffer[10];
 strcpy(buffer, s); // copy s into buffer
}

void main( int argc,  char** argv) {
  vulnerable(argv[1]);
        // argv[1] is first command line argument
}
```

What can go wrong here?

Buffer overflow in **`strcpy`** may corrupt the stack, with user input

Typical string problem: using **`gets`**

```
int main(int argc, char** archv i){
          char *msg = "hello";
          f();
          printf("%s", msg);
}
int f(){  char p[20];
          int j;
          gets(p);
          return 1;
         }
```

What can go wrong here?

`gets` reads user input until the first NULL character.

The program has no way of knowing how long this string will be.

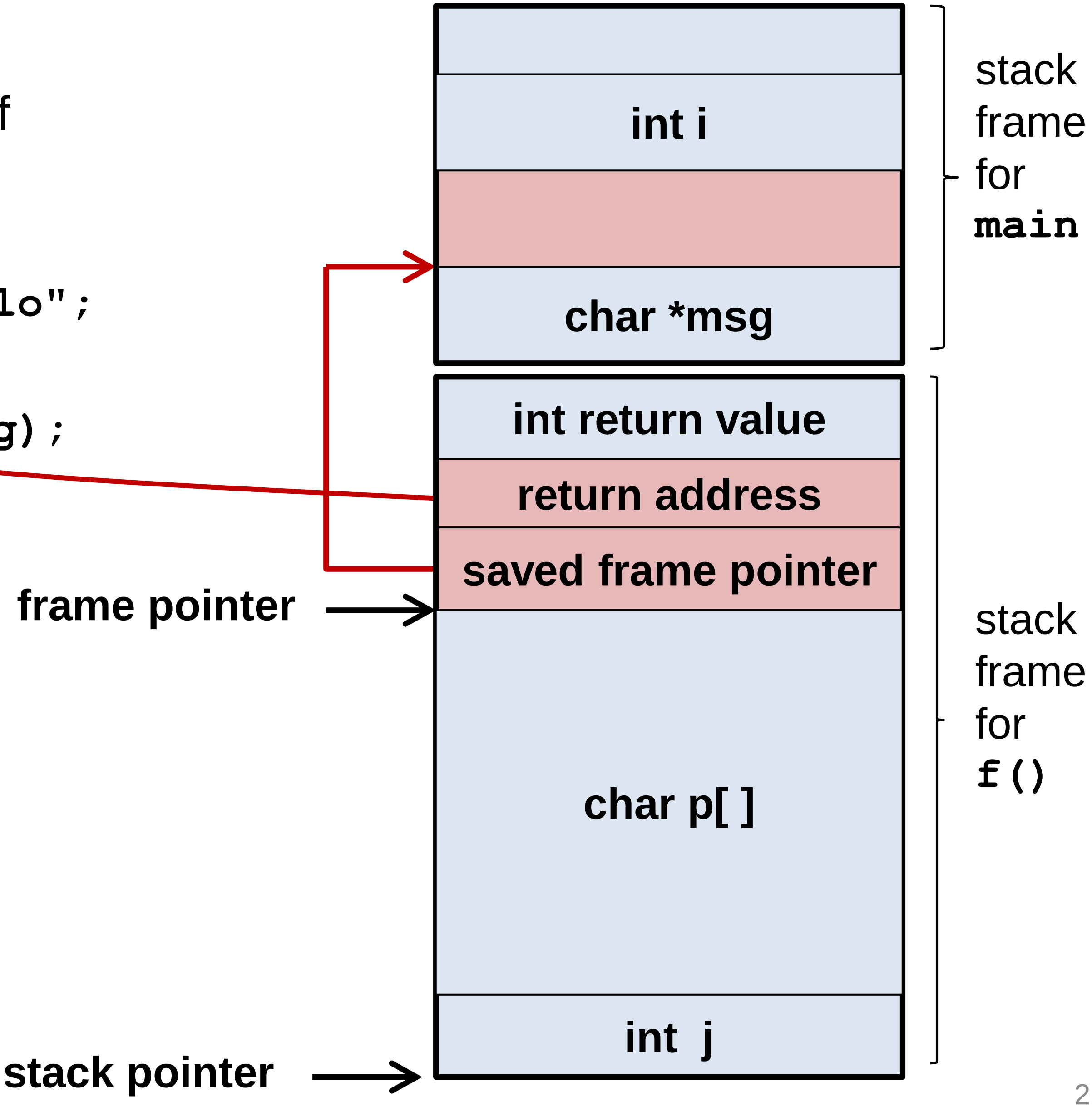
The stack can be corrupted with user input!

recall: the stack

Stack during call to f

```
main(int i){
 char *msg ="hello";
 f();
 print ("%s", msg);
}

int f(){
  char p[20];
  int j;
  gets(p);
  return 1;
}
```



int i

char *msg

stack frame for **`main`**

int return value

return address

saved frame pointer

char p[ ]

int j

stack frame for **`f()`**

frame pointer

stack pointer

Corrupting the stack (1)

What if we overrun ***`p`***

and to set return address

to point to some existing code,

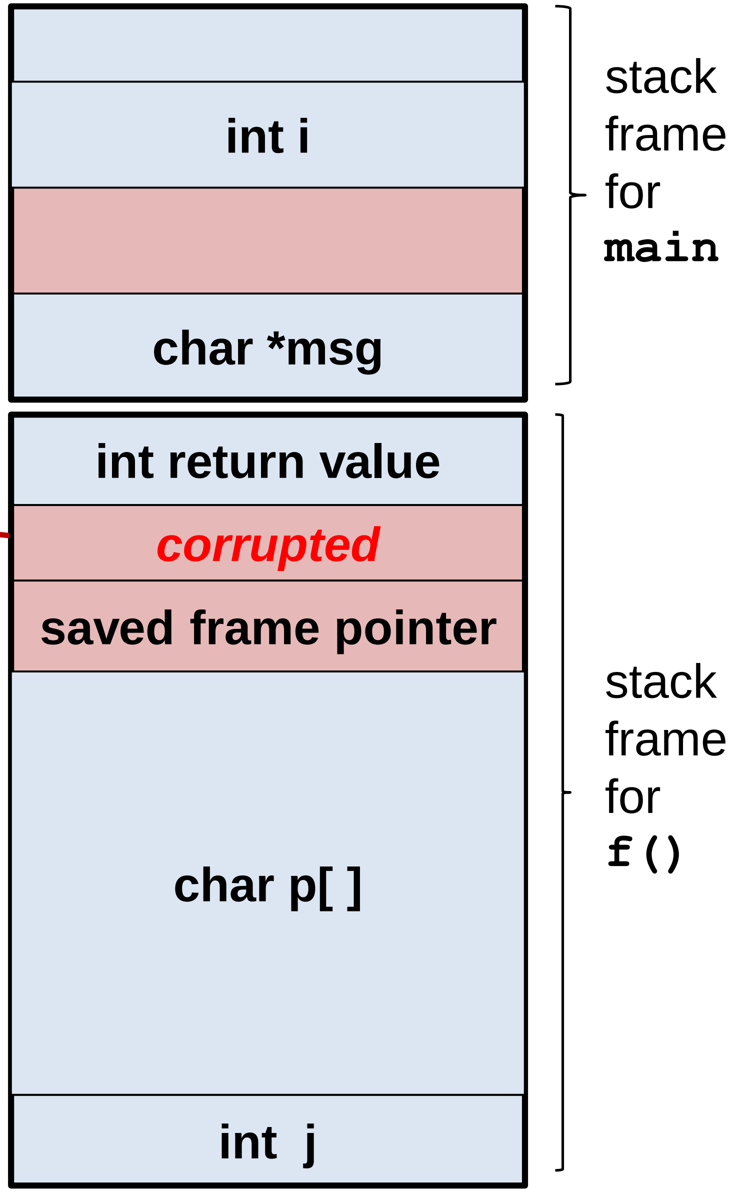
say inside a function ***`g()`****?*

When **`f`** returns,

execution will resume

with executing **`g`** instead

of **`main`**



int i

char *msg

stack frame for **`main`**

int return value

corrupted

saved frame pointer

char p[ ]

int j

stack frame for **`f()`**

Corrupting the stack (2)

What if we overrun ***`p`***

to set return address

to point inside p?

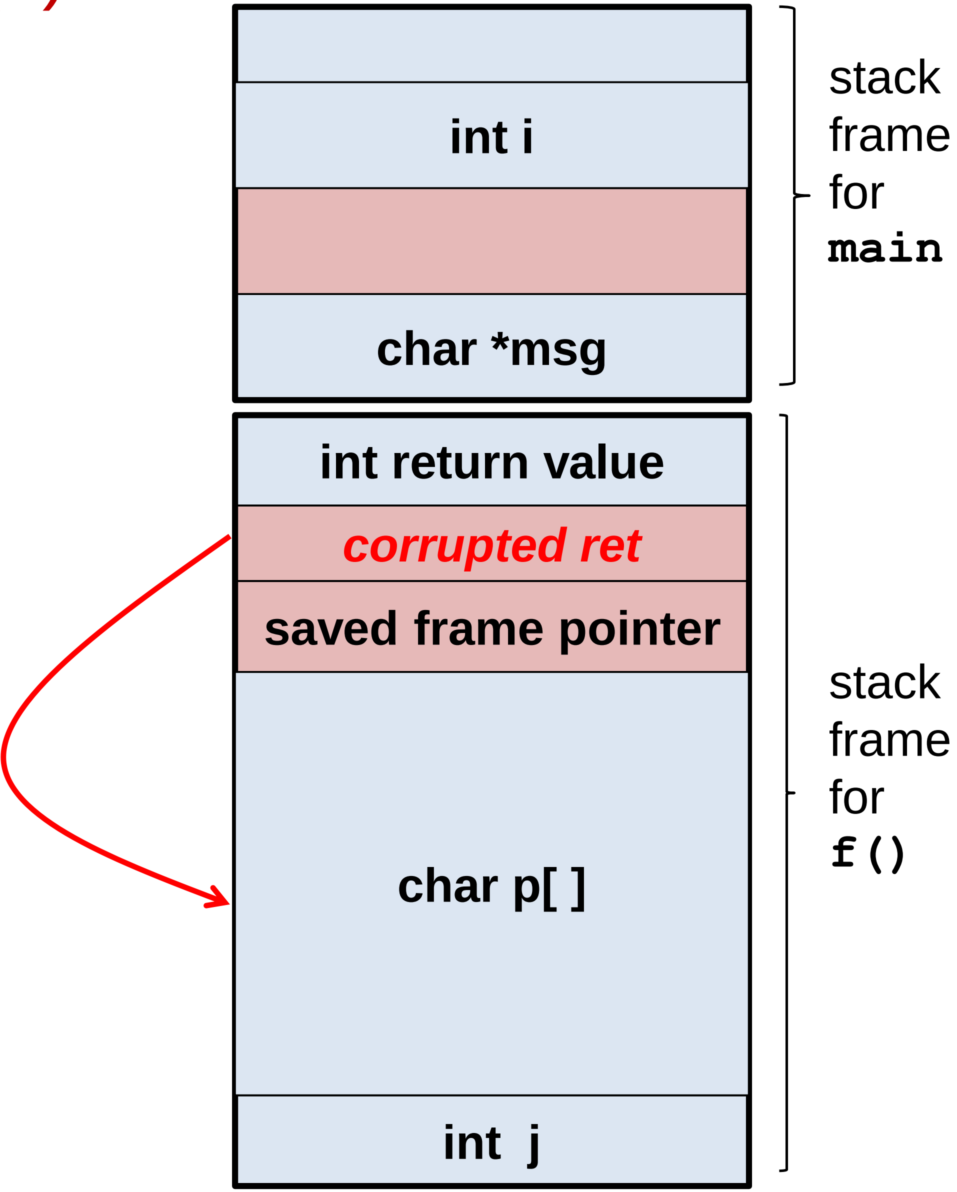
When **`f`** returns,

execution will resume

with what is written in **`p`**,

interpreted as machine

code



Corrupting the stack (3)

What if we overrun ***`p`***

to set saved frame pointer

to point inside p?

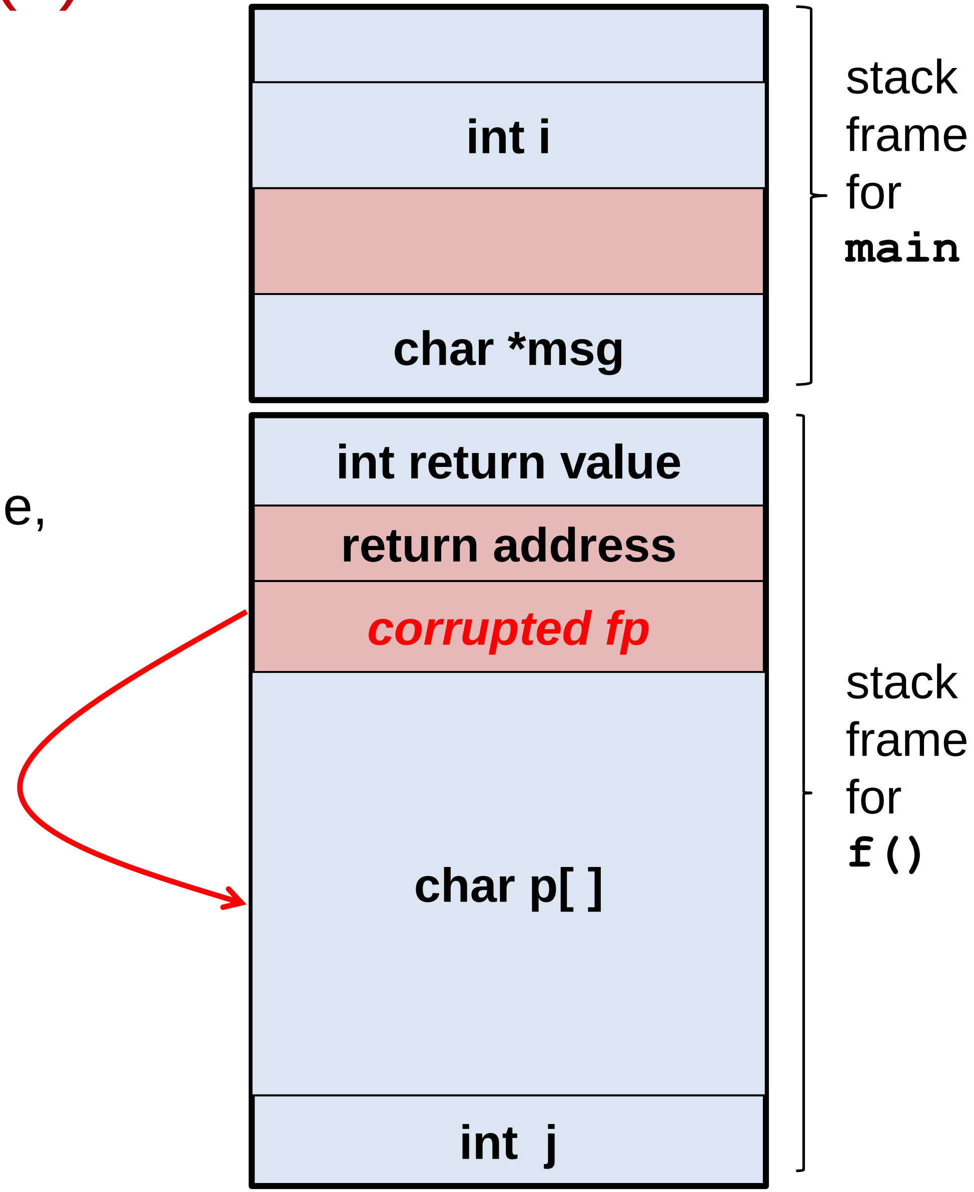
When **`f`** returns,

execution of **`main`** will resume,

but interpreting wrong part

of the stack as stack frame

for **`main`**



Corrupting the stack (4)

What if we overrun ***`p`***

and to set return address

to point to some existing code,

say inside a function ***`g()`**,*

and *to set saved frame*

pointer to point inside p?

When **`f`** returns,

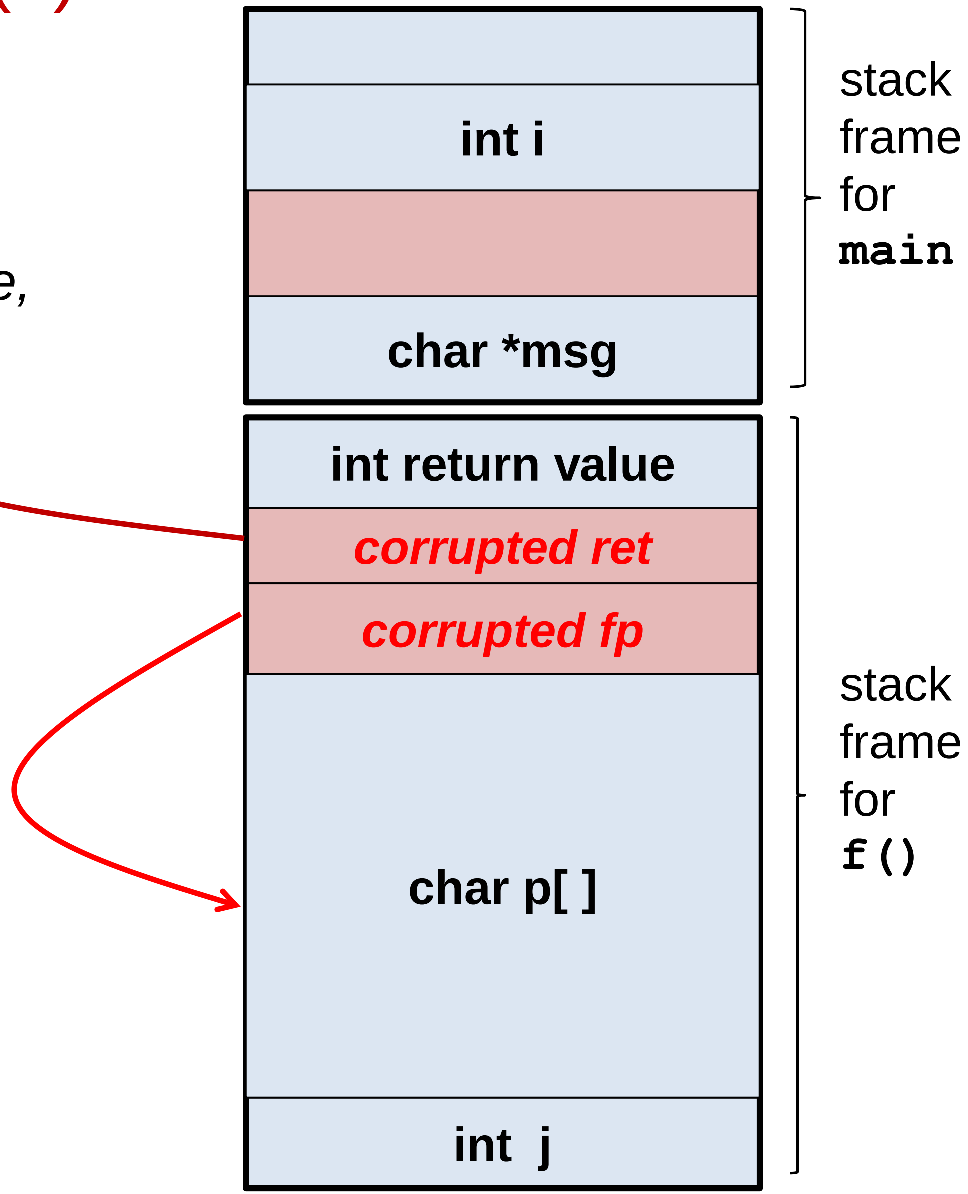
execution will resume

with executing **`g`** instead

of **`main`** and

interpreting stack starting

at **`p`** as a stack frame for ***`g`***



int i

char *msg

stack frame for **`main`**

int return value

corrupted ret

corrupted fp

char p[ ]

int j

stack frame for **`f()`**

Buffer overflow to change a program

Can attacker do something more interesting than crashing?

Yes, supplying a value for ret which will do something interesting

There are two possibilities for the attacker:

1. jumping to his own attack code (aka shell code)
   The attacker writes some program code into a buffer,
   and sets the return address to point to this code
2. jumping to some existing code, but with a malicious stack frame
   The attacker writes a fake stack frame into a buffer,
   and sets the return address to point to some existing code,
   and sets the saved frame pointer to point to this fake stack frame

NB lots of tricky details to get right!

pros & cons of where to jump

1. Jumping to own attack code (the original form of buffer overflow)

   CON: the attacker needs to know the address of the buffer

   CON: the memory page containing the buffer must be executable;
   on many modern systems the stack is not executable

2. Jumping to existing code with a manipulated stack frame

   PRO: does not require an executable stack
   or access to executable memory somewhere else

   CON: need to find the right code, and
   one or more fake frames must be put on the stack

   Often attacker will jump to functions in standard libc library,
   in so-called return-to-libc attack.

Both require the attacker to control the content of some buffers and corrupt the return address and frame pointer on the stack.

Shell code

Shell code

- If attacker manipulates the return address to jump to his own code, he needs some interesting code to jump to

- This code is known as shell code.
  - Traditionally, the goal is to spawn a shell, hence the name "shell code"

- The actual attack will involve
  1. somehow getting this shell code somewhere in memory
  2. overwriting the return address on the stack to this place where the shell code is

- The attacker can then do ***anything***

  within the rights & permissions of the program that is attacked.

How to spawn a shell

```
void main(int argc, char **argv) {
    char *name[2];
    name[0] = “/bin/sh“;
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```

course repository

How to spawn a shell

```
void main(int argc, char **argv) {
    char *name[2];
    name[0] = “/bin/sh“;
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```

```
(gdb) disas execve
  ....
 mov     0x8(%ebp),%ebx
 mov     0xc(%ebp),%ecx
 mov     0x10(%ebp),%edx
 mov     $0xb,%eax
 int     $0x80
  ....
```

How to spawn a shell

`int execve(char *file, char *argv[], char *env[])`

```
(gdb) disas execve
  ....
 mov      0x8(%ebp),%ebx
 mov      0xc(%ebp),%ecx
 mov      0x10(%ebp),%edx
 mov      $0xb,%eax
 int      $0x80
  ....
```

copy **file* to **`ebx`**

copy **argv[]* to **`ecx`**

copy **env[]* to **`edx`**

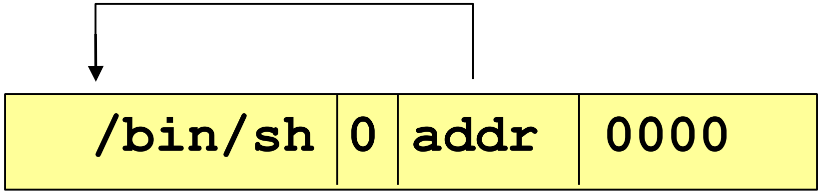
put the syscall number in eax
 (execve is **`0xb`**)

invoke the syscall

How to spawn a shell

Three parameters are needed

- **`*file`**: a null-terminated string **`\bin\sh`** somewhere in memory
- **`*argv[]`**: the address of that string **`\bin\sh`** followed by NULL (0x00000000)
- **`*env[]`**: some NULL in memory



The address problem: where am I?

- How can we put the address of the string **`\bin\sh`** in memory, if we do not even know where the position of the shellcode is?

- Trick to solve that
  - the CALL instruction puts the return address on the stack
  - if we put a CALL instruction just before the string **`\bin\sh`**, when it is executed it will push the address of the string onto the stack

The Shellcode (almost ready)

```
jmp     0x26                      # 2 bytes
popl    %esi                      # 1 byte
movl    %esi,0x8(%esi)            # 3 bytes
movb    $0x0,0x7(%esi)            # 4 bytes
movl    $0x0,0xc(%esi)            # 7 bytes
movl    $0xb,%eax                 # 5 bytes
movl    %esi,%ebx                 # 2 bytes
leal    0x8(%esi),%ecx            # 3 bytes
leal    0xc(%esi),%edx            # 3 bytes
int     $0x80                     # 2 bytes
movl    $0x1, %eax                # 5 bytes
movl    $0x0, %ebx                # 5 bytes
int     $0x80                     # 2 bytes
call    -0x2b                     # 5 bytes
.string \"/bin/sh\"               # 8 bytes
```

setup

execve()

exit()

setup

The zeroes problem

The shellcode is usually copied into a string buffer

```
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";
```

- Problem: the null byte **`\x00`** is the string terminator character which will stop any copying
- Solution: substitute any instruction containing zeros, with an alternative instruction

```
mov 0x0, reg --> xor reg, reg
mov 0x1, reg --> xor reg, reg
                 inc reg
```

The zeroes problem

- Some tools provide this functionality automatically:
  e.g., **`msfencode`** (metasploit framework)

Jumping into the buffer

- The buffer that we are overflowing is usually a good place to put the code (shellcode) that we want to execute

- The buffer is somewhere on the stack, but in most cases the exact address is unknown
  - the address must be precise: jumping one byte before or after would just make the application crash
  - on the local system it is possible to find out the address with a debugger, but it is very unlikely to be exactly the same address on a different machine
  - any change to the environment variables affects the stack position

Solution 1: the NOP sled

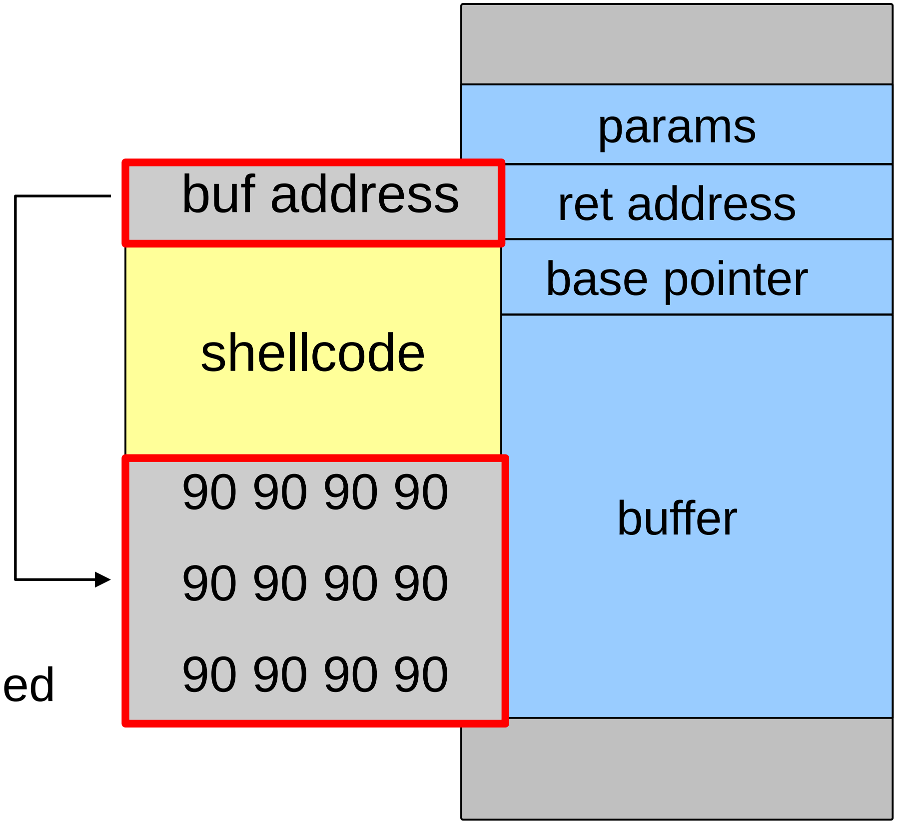
- A sled is a “landing area” that is put in front of the shellcode

- The simplest sled is a sequence of no operation (NOP) instructions
  - NOP is a single byte instruction (0x90) that does not do anything

  If the program jumps anywhere into the NOP sled, with will execute these NOPs and then the shell code

- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area

Assembling the malicious buffer



The NOP sled

Solution 2: jump using a register

- Find a register that points to the buffer (or somewhere into it)
  - ESP
  - EAX (return value of a function call)

- Locate an instruction that jumps/calls using that register
  - can also be in one of the libraries
  - does not even need to be a real instruction, just look for the right sequence of bytes

- Overwrite the return address with the address of that instruction

Recap

Recap

An attacker feeding malicious input to insecure code can

1. leak data
2. corrupt data
3. change program execution entirely

This can happen due to buffer overflows or format string attacks

When using buffer overflows to change program behaviour
an attacker can

1. inject his own code or
2. jump to existing code with a fake stack frame

More general trends

Format string problems are easy to fix,

eg replacing **`printf(msg)`**

by **`printf(”%s”, msg)`**

(for all functions of the **`*printf`** family!)

and are then no longer a threat.

Still, they are a representative of many examples where some small feature in one function can be a source of security vulnerabilities

- Such vulnerabilities typically involve *special characters* which are *interpreted* in a special way at runtime
- Note that this means that such characters are effectively more like *program code* than just *data*

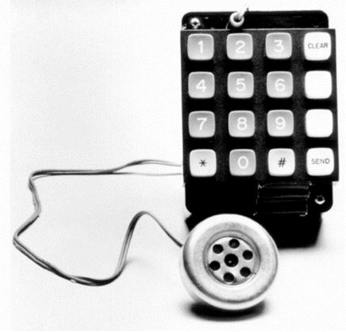
pre-history of hacking

In 1950s, Joe Engressia showed the telephone network could be hacked by phone phreaking:

ie. whistling at right frequencies

http://www.youtube.com/watch?v=vVZm7I1CTBs

In 1970s, before founding Apple together with Steve Jobs, Steve Wozniak sold Blue Boxes for phone phreaking at university



Common theme: mixing channels

The root cause of phone phreaking & buffer overflows is the same!

- signals to control the telephone switchboards *(beeps at certain frequencies)* are sent over the same channel as untrusted user data (*the phone calls*)
- data to control execution *(return addresses)* are stored in the same places as untrusted user data (*user input*)

These *attack vectors* give the attacker control over the phone network and the computer, respectively

Common theme: mixing channels

Moral of the story:

- Don’t mix data and code!

  Here data is phone call or user input,

  code is control beeps or return addresses

- History repeats itself!

  Not just phone phreaking & stack overflow,

  but also XSS, SQL injection, OS command injection, …