First of all, what’s a buffer?

- **Buffer** is a continuous block of memory that holds multiple instances of the same data type
  - E.g. an array of characters
- **Arrays in C** can be allocated in several ways:
  - Statically, allocated at load time
  - Dynamically, allocated at run time on the program stack
    - The type we are interested in here
  - Dynamically, allocated at run time on the heap
What’s a buffer overflow?

- Writing past the end of a buffer

  E.g.:
  ```
  char large_string[20];
  int i;
  for(i = 0; i < 20; i++)
    large_string[i] = 'A';
  char buffer[16];
  char next_buffer[4];
  strcpy(buffer,large_string);
  ```

- Used for **stack smashing** attacks

Organization of memory for a process

- **Text** region contains code and read-only data
  - Marked as read-only; write attempts result in segmentation faults
- **Initialized and uninitialized data** region contains static variables
- **Stack** is convenient to implement subprogram calls
  - Stack frames are used to allocate by-value parameters and local variables dynamically
  - A stack pointer (SP) register points to the top of the stack
  - The bottom of the stack is fixed
Lecture 3: Buffer overflow and race condition exploits

Example where buffer overflow causes segmentation fault

```c
#include <stdio.h>

void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
```

Stack frame allocated for the call to function:

<table>
<thead>
<tr>
<th>buffer</th>
<th>fp</th>
<th>ip</th>
<th>str*</th>
</tr>
</thead>
</table>

Example where stack manipulations are used to change execution of the program

```c
#include <stdio.h>

void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    int *ret; ret = buffer1 + 12;
    (*ret) += 10;
}

void main() {
    int x; x = 0;
    function(1,2,3);
    x = 1;
    printf("%d\n",x);
}
```

Stack frame allocated for the call to function:

<table>
<thead>
<tr>
<th>buffer2</th>
<th>buffer1</th>
<th>fp</th>
<th>ip</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
Lecture 3: Buffer overflow and race condition exploits

How did we know to add 10 bytes to the return address?

- Debuggers often come handy... From running gdb on the program:

```bash
$ gdb example3
GDB is free software and you are welcome to distribute copies of it under certain conditions; type "show copying" to see the conditions. There is absolutely no warranty for GDB; type "show warranty" for details. GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc... (no debugging symbols found)...
```

Dump of assembler code for function main:
```
0x8000490 <main>: pushl %ebp
0x8000491 <main+1>: movl %esp,%ebp
0x8000493 <main+3>: subl $0x4,%esp
0x8000496 <main+6>: movl $0x0,0xfffffffc(%ebp)
0x8000499 <main+9>: pushl %ebp
0x800049f <main+15>: pushl %ebp
0x80004a1 <main+17>: pushl %ebp
0x80004a3 <main+19>: call 0x8000470 <function>
0x80004a8 <main+24>: addl $0xc,%esp
0x80004ab <main+27>: movl $0x1,0xfffffffc(%ebp)
0x80004b1 <main+34>: movl 0xfffffffc(%ebp),%eax
0x80004b5 <main+37>: pushl %eax
0x80004b6 <main+38>: pushl 0x80004f8
0x80004bf <main+43>: call 0x8000378 <printf>
0x80004c0 <main+48>: addl $0x8,%esp
0x80004c3 <main+51>: movl %ebp,%esp
0x80004c5 <main+53>: popl %ebp
0x80004c6 <main+54>: ret
```

OK, but what if we want to force the program to do something very specific?

- Simple --- fill a program with code you want executed and set instruction pointer to the beginning of this code.
What code do we write there?

- How about spawning a shell?
- Write a C program:

```c
#include <stdio.h>
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- Compile and run `gdb` to get machine instructions

A buffer overflow must exist in a program and be exploitable for an attacker to take advantage of the stack smashing attack

- The best defense is to avoid buffer overflows
  - Use a high-level language that does not allow pointer arithmetic
  - Avoid certain functions that can overflow buffers or insert explicit checks
    - `strcpy` copies all characters from the source buffer to the destination buffer, without checking sizes
      - Insert checks based on `strlen`
      - Use `strncpy`
    - `gets` reads user test until an end-of-file or newline character
      - Use `fgets` instead
Race conditions

- **Definition:** A race condition is anomalous behavior caused by the unexpected dependence on the relative timing of events. In other words, a programmer incorrectly assumed that a particular event would always happen before another.
- **A typical example:** a reader-writer problem
  - A number of threads write to and read from a shared buffer.
  - Each value is supposed to be written and read only once, in a FIFO fashion.
  - A race condition occurs if two writer threads access the buffer at the same time and only one value is written.
- **See Banking example**

So, what is happening in the Banking example?

```
Spender
withdraw 1000
newAmount=9000
amount=10000
newAmount=8000
withdraw 1000
newAmount=7000

Account
newAmount=9000
amount=10000
newAmount=8000
amount=9000
newAmount=7000

Earned
newAmount=12000
deposit 1000
newAmount=11000
return
```
Protecting data through synchronized methods

- Any synchronized method can only be executed by one thread at a time
  - All synchronized methods for an object comprise a monitor

```java
public class Account {
    // attributes

    public synchronized void deposit(int amount) {
        // as before
    }

    public synchronized void withdraw(int amount) {
        // as before
    }

    public synchronized int getBalance() {
        // as before
    }
}
```

Only one thread can execute any of `deposit`, `withdraw`, and `getBalance` at any given time.

What do the other threads do?

- If a thread `t1` has to execute a synchronized method of some object and some other thread is already executing a synchronized method of that object, `t1` releases its resources and waits until the other thread is done.

- The language does not define the order in which threads get access to synchronized resources
  - Does not have to be first in, first out order
Synchronizing methods is not the only mechanism for managing access by threads in Java
• What if an account in the Banking Account example is overdrawn?

Using the wait() method to temporarily suspend a thread executing in a monitor
• A thread may suspend itself by calling the wait() method of the lock object
  ○ If no object is specified, the this object is used
  ○ Calls to wait() cannot occur outside synchronized regions or methods
    A runtime exception is thrown
  ○ After the call, the thread leaves the monitor
    Other threads may execute in this monitor
• Back to the example
  ○ Make the thread executing the withdraw() method of Account wait if the account is about to be overdrawn
Lecture 3: Buffer overflow and race condition exploits

A note aside: **deadlocks**

- A deadlock is a situation where a number of threads cannot continue because they wait on some action from each other
  - In the Bank Account example:
    - The earner thread successfully terminates
    - The spender thread is suspended after `wait()`
    - The main thread is waiting for the withdrawer thread to complete (executing the `join()` method)
    - The program is permanently locked

- Deadlocks are always bad and should be avoided

---

Reviving suspended threads

- After a thread executes `wait()`, something should un-suspend it at some point
  - Done by some other thread calling either `notify()` or `notifyAll()` method of the lock object
    - `notify()` selects (arbitrarily!) one of potentially many waiting threads and un-suspends it
    - `notifyAll()` un-suspends all waiting threads for the given lock object

- Back to the example
  - Make the thread executing the `deposit()` method of `Account` un-suspend the thread that may be waiting
When a thread is notified, how does it proceed?

- The notified thread has to resume its execution
- But it has to do so in a monitor
  - Some other threads may be executing in this monitor
- The notified thread waits until there are no other threads in the monitor
  - Just as if it was trying to execute a synchronized region from the start

OK, so how is this related to security?

- The goal of an attacker is to do something in parallel with a running program to force it to do something bad
- The most common variety: time-of-check, time-of-use (TOCTOU) exploits
  - The program checks some security condition before using some security sensitive resources
  - The attacker lets it do a check and then hijacks the resources
Lecture 3: Buffer overflow and race condition exploits

**TOCTOU example: broken passwd program on SunOS**
- passwd changes a password for the user running the program
- Takes the password file as input
- Performs 4 steps:
  1. Open the password file and retrieve the entry for the user running the program
  2. Create and open a temp file called ptmp in the same directory as the password file
  3. Open the password file and copy the unchanged contents into ptmp; write the changed password
  4. Close both files, then rename ptmp to be the password file

**The attack**

passwd program

attacker

$ cd attack-dir
$ mkdir pwd
$ touch pwd/.rhosts
$ echo "localhost attacker :::::" >> pwd/.rhosts
$ ln –s target-dir link
$ passwd link/.rhosts

$ rm link
$ ln –s target-dir link
$ in –s target-dir link

Open attack-dir/pwd/.rhosts, read the entry for the attacker
Create and open a file ptmp in target-dir
Open attack-dir/pwd/.rhosts, copy the unchanged data in target-dir/ptmp
Close attack-dir/pwd/.rhosts and target-dir/ptmp
Copy target-dir/ptmp to target-dir/.rhosts
Exit

Login as the user who owns target-dir (root?), without a password
Lecture 3: Buffer overflow and race condition exploits

Avoiding TOCTOU attacks

- Avoid file system calls that take file names as inputs
  - Use file handles instead
- Avoid using access call on files
  - Checks if the process running the program has permission to access the file
- Be careful when using temp files
  - Attackers may be able to guess their names
- ...

Not all race condition attacks use the file system accesses

- In the Auction program:
  - What happens if several users bid on the same auction at roughly the same time?
  - What happens if an administrator removes an auction and a user bids on this auction at roughly the same time?
  - ...

Page <#>