# Last time

- What is our goal in this course?
- What is security?
- What is privacy?
- Who are the adversaries?
- Assets, vulnerabilities, threats, attacks and controls
- Methods of defence

# This time

- Program security
- Flaws, faults, and failures
- Types of security flaws
- Unintentional security flaws
  - Buffer overflows
  - Incomplete mediation
  - TOCTTOU errors

# Secure programs

- Why is it so hard to write secure programs?
- A simple answer [CHE02]:
  - Axiom (Murphy):
    Programs have bugs
  - Corollary:
    Security-relevant programs have security bugs

# Flaws, faults, and failures

- A flaw is a problem with a program
- A security flaw is a problem that affects security in some way
  - Confidentiality, integrity, availability
- Flaws come in two types: faults and failures
- A fault is a mistake "behind the scenes"
  - An error in the code, data, specification, process, etc.
  - A fault is a *potential* problem

# Flaws, faults, and failures

- A failure is when something actually goes wrong
  - You log in to the library's web site, and it shows you someone else's account
  - “Goes wrong” means a deviation from the desired behaviour, not necessarily from the specified behaviour!
    - The specification itself may be wrong
- A fault is the programmer/specifier/inside view
- A failure is the user/outside view

# Finding and fixing faults

- How do you find a fault?
  - If a user experiences a failure, you can try to work backwards to uncover the underlying fault
  - What about faults that haven't (yet) led to failures?
  - Intentionally try to *cause* failures, then proceed as above
    - Remember to think like an attacker!
- Once you find some faults, fix them
  - Usually by making small edits (called patches) to the program
  - This is called “penetrate and patch”
  - Microsoft's “Patch Tuesday” is a well-known example

# Problems with patching

- Patching sometimes makes things *worse*!
- Why?
  - Pressure to patch a fault is often high, causing a narrow focus on the observed failure, instead of a broad look at what may be a more serious underlying problem
  - The fault may have caused other, unnoticed failures, and a partial fix may cause incosistencies or other problems
  - The patch for this fault may introduce new faults, here or elsewhere!
- Alternatives to patching?

# Unexpected behaviour

- When a program's behaviour is speficied, the spec usually lists the things the program must do
  - The ls command must list the names of the files in the directory whose name is given on the command line, if the user has permissions to read that directory
- Most implementors wouldn't care if it did additional things as well
  - Sorting the list of filenames alphabetically before outputting them is fine

# Unexpected behaviour

- But from a security / privacy point of view, extra behaviours could be bad!
  - After displaying the filenames, post the list to a public web site
  - After displaying the filenames, delete the files
- When implementing a security or privacy relevant program, you should consider “and nothing else” to be implicity added to the spec
  - “should do” vs. “shouldn't do”
  - How would you test for “shouldn't do”?

# Types of security flaws

- One way to divide up security flaws is by genesis (where they came from)
- Some flaws are intentional
  - Malicious flaws are intentionally inserted to attack systems, either in general, or certain systems in particular
    - If it's meant to attack some particular system, we call it a targeted malicious flaw
  - Nonmalicious (but intentional) flaws are often features that are meant to be in the system, and are correctly implemented, but nonetheless can cause a failure when used by an attacker
  - We will look at intentional flaws in the next two lectures

# Unintentional program errors

- Most security flaws are caused by unintentional program errors
- For the rest of this lecture, we will look at some of the most common sources of unintentional security flaws
  - Buffer overflows
  - Incomplete mediation
  - TOCTTOU errors (race conditions)

# Buffer overflows

- The single most commonly exploited type of security flaw
- Simple example:

```
#define LINELEN 1024

char buffer[LINELEN];

gets(buffer);
```

*or*

```
strcpy(buffer, argv[1]);
```

# What's the problem?

- The `gets` and `strcpy` functions don't check that the string they're copying into the buffer will fit in the buffer!

- So?

- Some languages would give you some kind of exception here, and crash the program.
  - Is this an OK solution?

- Not C (the most commonly used language for systems programming). C doesn't even notice something bad happened, and continues on its merry way.

# Smashing The Stack For Fun And Profit

- This is a classic (read: somewhat dated) exposition of how buffer overflow attacks work.

- Upshot: if the attacker can write data past the end of an array on the stack, she can usually overwrite things like the saved return address. When the function returns, it will jump to any address of her choosing.

- Targets: programs on a local machine that run with setuid (superuser) privileges, or network daemons on a remote machine

# Kinds of buffer overflows

- In addition to the classic attack which overflows a buffer on the stack to jump to shellcode, there are many variants:
  - Attacks which work when a single byte can be written past the end of the buffer (often caused by a common off-by-one error)
  - Overflows of buffers on the heap instead of the stack
  - Jump to other parts of the program, or parts of standard libraries, instead of shellcode

# Defences against buffer overflows

- How might one protect against buffer overflows?
- Use a language with bounds checking
  - And catch those exceptions!
- Non-executable stack
- Stack (and sometimes code) at random addresses for each process
  - Linux 2.6 does this
- “Canaries” that detect if the stack has been overwritten before the return from each function
  - This is a compiler feature

# Incomplete mediation

- Inputs to programs are often specified by untrusted users
  - Web-based applications are a common example
- “Untrusted” to do what?
- Users sometimes mistype data in web forms
  - Phone number: `51998884567`
  - Email: `iang#cs.uwaterloo.ca`
- The web application needs to ensure that what the user has entered constitutes a meaningful request
- This is called mediation

# Incomplete mediation

- Incomplete mediation occurs when the application accepts incorrect data from the user
- Sometimes this is hard to avoid
  - Phone number: `519-886-4567`
  - This is a reasonable entry, that happens to be wrong
- We focus on catching entries that are clearly wrong
  - Not well formed
    - DOB: `1980-04-31`
  - Unreasonable values
    - DOB: `1876-10-12`
  - Inconsistent with other entries

# Why do we care?

- What's the security issue here?
- What happens if someone fills in:
  - DOB: `9876487423649248364924783648923649 2`
    - Buffer overflow?
  - DOB: `'; DROP DATABASE clients --`
    - SQL injection?
- We need to make sure that any user-supplied input falls within well-specified values, known to be safe

# Client-side mediation

- You've probably visited web site with forms that do *client-side* mediation
  - When you click “submit”, Javascript code will first run validation checks on the data you entered
  - If you enter invalid data, a popup will prevent you from submitting it
- Related issue: client-side state
  - Many web sites rely on the client to keep state for them
  - They will put hidden fields in the form which are passed back to the server when the user submits the form

# Client-side mediation

- Problem: what if the user
  - Turns off Javascript?
  - Edits the form before submitting it? (Greasemonkey)
  - Writes a script that interacts with the web server instead of using a web browser at all?
  - Connects to the server “manually”? (telnet server.com 80)
- Note that the user can send arbitrary (unmediated) values to the server this way
- The user can also modify any client-side state

# Example

- At a bookstore website, the user orders a copy of the course text. The server replies with a form asking the address to ship to. This form has hidden fields storing the user's order
  - <input type="hidden" name="isbn"
    value="0-13-239077-9">
    <input type="hidden" name="quantity"
    value="1">
    <input type="hidden" name="unitprice"
    value="111.00">
- What happens if the user changes the "unitprice" value to "50.00" before submitting the form?

# Defences against incomplete mediation

- Client-side mediation is an OK method to use in order to have a friendlier user interface, but is useless for security purposes.
- You have to do server-side mediation, whether or not you also do client-side.
- For values entered by the user:
  - Always do very careful checks on the values of all fields
  - These values can potentially contain completely arbitrary 8-bit data (including accented chars, control chars, etc.) and be of any length
- For state stored by the client:
  - Make sure the client has not modified the data in any way

# TOCTTOU errors

- TOCTTOU ("TOCK-too") errors
  - Time-Of-Check To Time-Of-Use
  - Also known as "race condition" errors

- These errors occur when the following happens:
  - User requests the system to perform an action
  - The system verifies the user is allowed to perform the action
  - The system performs the action

# Example

- A particular Unix terminal program is setuid (runs with superuser privileges) so that it can allocate terminals to users (a privileged operation)
- It supports a command to write the contents of the terminal to a log file
- It first checks if the user has permissions to write to the requested file; if so, it opens the file for writing
- The attacker makes a symbolic link:
  `logfile -> file_she_owns`
- Between the “check” and the “open”, she changes it:
  `logfile -> /etc/passwd`

# The problem

- The state of the system changed between the check for permission and the execution of the operation

- The file whose permissions were checked for writeability by the user (`file_she_owns`) wasn't the same file that was later written to (`/etc/passwd`)
  - Even though they had the same name (`logfile`) at different points in time

- Q: Can the attacker really “win this race”?
- A: Yes.

# Recap

- Program security
- Flaws, faults, and failures
- Types of security flaws
- Unintentional security flaws
  - Buffer overflows
  - Incomplete mediation
  - TOCTTOU errors

# Next time

- Finish TOCTTOU
- Malicious code: Malware
  - Viruses
  - Trojan horses
  - Logic bombs
  - Worms
- Other malicious code: web bugs