Security - Why Bother?

• Your projects in this class are not likely to be used for some critical infrastructure or real-world sensitive data. Why should you be concerned about security?
Security - Why Bother?

• Most secure coding practices stem from the following principle:
  – For an attacker to develop an exploit against your project, some bug in your code must form the basis for that attack.
  – Bugs => exploits
  – Bugs => programmer headaches
  – No bugs => fewer exploits and fewer headaches
Security - Why Bother?

• The headaches associated with debugging grow along with the project size and complexity.
  – Very small errors can become buried in a large project, and can be difficult to find later.
  – Secure programming practices make bugs easier to find and less likely to occur.
Security – Why Bother?

• People just like you write software that is used in the real world and it gets hacked all the time.
• Don’t be that guy!
Common Security Terms

• Exposure
  – Possible loss or harm in a computing system

• Vulnerability
  – A weakness in a computer-based system that may be exploited to cause loss or harm

• Attack
  – An exploitation of a system vulnerability

• Threats
  – Circumstances that have potential to cause loss or harm

• Control
  – A protective measure that reduces a system vulnerability
Types of damage from External Attack

• Denial of Service
  – Normal usage of the system by authorized users has been denied

• Corruption of programs or data
  – Program or data contents can be modified, producing undesirable results

• Disclosure of confidential information
  – Including things like passwords or SSN
To Ensure Security of a System

1. Vulnerability Avoidance
   – i.e. don’t connect a computer to the network if it doesn’t need to communicate with others

2. Attack Detection & Neutralization
   – i.e. using a virus checker to protect the computer from further exploitation

3. Exposure Limitation
   – i.e. regular back ups and constant security patch application protect a system from long term exposure
Dependable Software

• Fault Avoidance
  – Design and implement the system in such a way as to minimize human error

• Fault Tolerance
  – Design and implement the system such that even if faults do occur, they do not negatively impact the system
Documentation

• Comments are important for security
  – People who run your code should know what to expect
    • Document preconditions, postconditions, error handling, semantics of usage.
  – People who examine and debug your code should know exactly how it works
    • Document your actual code so that others can follow it.
Documentation

• Readability is key, both for comments and code.
  – Whitespace
  – Indentation

• Always keep in mind the idea that someone else might have to maintain your code after you. Think golden rule!
Checking Input and Bounds

• Many errors stem from not validating user input, or not checking array bounds
  – These problems lead to classic exploit types, such as the stack smashing attacks
  – Some languages, like Java, will perform some bounds-checking automatically, but it is still a good idea to handle it yourself.
    • Write your own checks
    • Handle exceptions generated by the language
Checking Input and Bounds

• DTA (Don’t Trust Anyone)
  – Users are not the only source of bad input
    • Open-source programs
    • Project components written by other group members
    • Project components written by you
  – If your code validates input and results from all external sources, errors will be less prevalent
    • Also, the source of an error will be much easier to track down.
  – Don’t go overboard with validation.
Network Aware Code Cautions

• Always be suspect of software designed to communicate over a network
  – More opportunities for outsiders to abuse your program
  – An error/bug in your code could negatively affect the network resources and other host computers
Pointers and Access Control

• Access control is a great feature of Object-Oriented Programming
  – Can lock down private data members
    • Security support from language
    • Excellent from a debugging standpoint
  – Pointers can violate this scheme
    • What happens if a client program gets a pointer to some private data of a class instance?
      – Security problem
      – Debugging nightmare
Pointers and Access Control

• Be careful with pointers
  – This is usually easy in C/C++
    • Pointers have special characters like *, &, and -> associated with them (most of the time)
  – Not necessarily so easy in Java
    • In Java, there are no pointers.
    • Non-primitive variables are really pointers, but there is no obvious distinction in your code.
Secure Programming Practices

• In short - try to expect all possible exceptional cases your code could encounter.
  – Handle them appropriately
  – Printing debugging information can be useful, but in your final revision, too much information can help adversaries work against you
  – Granting root-level access is not an appropriate exception handling policy.
Good Testing Reduces Dangers

• Requirements inspection & management
  – Continually review your requirements to make sure that you are developing your code to spec

• Code inspections
  – Have your peers review your code looking for common problems and ask why you designed certain things in certain ways

• Static analysis
  – Run a tool like lint or ITS4 against your code to search for common programming errors
  – See http://www.wiretapped.net/indexes/development.html for more information and other tools

• Good Test planning & management
  – Ensures that you cover all the bases and ensures that system requirements match up well with the final product
Static Analysis

• Software tool which scans the source text of a program and detects possible faults or anomalies
• Frequently used during code inspections
• Can be integrated into your development environment
  – ITS4 can be melded with emacs
  – /GS is the new part of MS Visual C++
Static Analysis Checks For…

• Control flow analysis
  – Highlights loops with multiple entry & exit points and unreachable code

• Data use analysis
  – Checks variable usage for things like
    • Variables that are declared but never used
    • Variables that are written twice without an intervening assignment
Static Analysis Checks For…

• Interface Analysis
  – Performs additional type checking to ensure proper use

• Information Flow Analysis
  – Identifies dependencies between input and output variables

• Path Analysis
  – Unravels your program to show all the possible paths