### CS 194-1 (CS 161) Computer Security

### Lecture 14

### Principles; Software security (defensive programming)

October 18, 2006 Prof. Anthony D. Joseph http://cs161.org/

### Review

Attackers will exploit any and all flaws!

Buffer overruns, format string usage errors, implicit casting, TOCTTOU, ...

Trusted Computing Base (TCB)

System portion(s) that must operate correctly for system security goals to be assured
Desired properties: Reference Monitor

Three Cryptographic principles

Conservative Design, Kerkhoff's Principle, Proactively Study Attacks

First two principles

Security is Economics, Least Priviledge

Lec 14.2

10/18/06 Joseph CS161 ©UCB Fall 2006

### Goals for Today

- · Principles for building secure systems
  - 11 other principles
  - Principles are neither necessary nor sufficient to ensure a secure system design, but they are often very helpful
  - Goal is to explore what you can do at design time to improve security
- Implementation techniques to avoid security holes when writing code
  - Several good practices

10/18/06

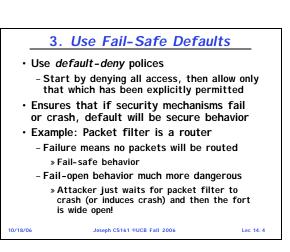
10/18/06

- Lots of overlap with software engineering and general software quality, but security places heavier demands

Lec 14.3

Lec 14.5

Joseph CS161 ©UCB Fall 2006

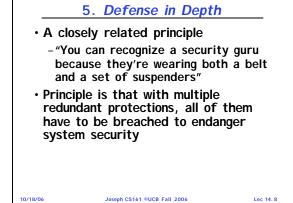


## Non-Fail-Safe Defaults Examples SunOS machines used to ship with + in /etc/hosts.equiv file Allowed anyone with root access on any machine on the Internet to log into your machine as root Irix machines used to ship with xhost + in their X Windows configuration files Allowed anyone to connect to Xserver

Joseph CS161 ©UCB Fall 2006

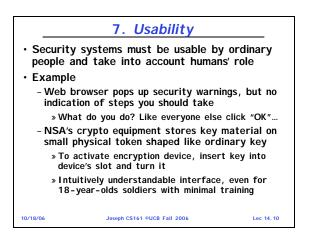
4. Separation of Responsibility	
• Split up privilege	
- No one person or program has complete power	
<ul> <li>Require more than one party to approve before access is granted</li> </ul>	
<ul> <li>Two-party rule examples</li> </ul>	
<ul> <li>Movie theater: pay teller and get ticket stub, then separate employee tears ticket in half, collects a half of it and puts it in lockbox</li> </ul>	
» Helps prevent insider fraud (under-/over-charge)	
<ul> <li>Most companies: purchases over certain amount must be approved by both requesting employee and a purchasing officer</li> </ul>	
» Helps prevent insider fraud in vendor choice	
10/18/06 Joseph CS161 ©UCB Fall 2006 Lec 14.6	





### 6. Psychological Acceptability

- Important that users buy into security model
  Examples
  - Company FW admin capriciously blocks apps that engineers need to get their jobs done » They view FW as damage and tunnel around it
  - Sys admin makes all passwords auto-generated long unmemorizable strings changed monthly
    - » Users simply write down their passwords on yellow post-its attached to their screens
- No system can remain secure for long when all its users actively seek to subvert it
  - Sys admins aren't going to win this game...
- Well-intentioned edicts can ultimately turn out to be counter-productive 10/18/06 Joseph CS161 9UCB Fall 2006 Lec 14.9

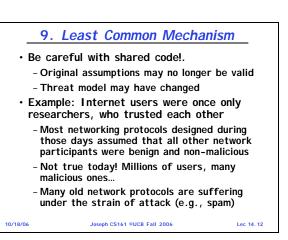


### 8. Ensure Complete Mediation

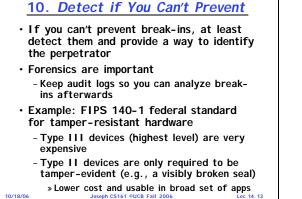
- When enforcing access control policies, ensure that *every* access to *every* object is checked
- Caching is a slightly sticky subject
  - Can sometimes avoid checking every access and allowing security decisions to be cached, but beware
- What if context relevant to security decision changes, and cache entry isn't invalidated?
  - Someone might get away with accessing something they shouldn't

10/18/06

Joseph CS161 ©UCB Fall 2006



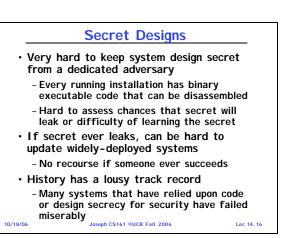
Lec 14, 11



10/18/0

### 11. Orthogonal Security We've seen this one before... Security mechanisms implemented orthogonally (transparently) to rest of system are useful in protecting legacy systems · Also, allow us to improve assurance by composing multiple mechanisms in series 10/18/06 Joseph CS161 ©UCB Fall 2006 Lec 14, 14

12. Don't Rely on Security Through Obs	scurity
• We've seen this one in the last lecture	e
<ul> <li>'Security through obscurity' phrase</li> </ul>	
<ul> <li>Systems that rely on secrecy of design algorithms, or source code to be secur</li> </ul>	
<ul> <li>Claimed reasoning:</li> </ul>	
<ul> <li>"This system is so obscure, only 100 people understand anything about it, s what are the odds that adversaries wi bother attacking it?"</li> </ul>	
<ul> <li>Self-defeating approach</li> </ul>	
- As system becomes more popular, more incentive to attack it, and cannot rely its obscurity to keep attackers away	



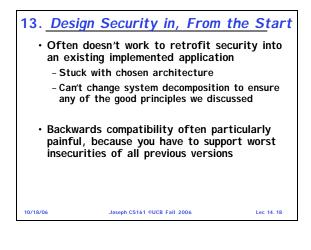
### What About Open Source? Are open-source applications more secure than closed-source applications?

- Not necessarily
- · Don't trust any system that relies on security through obscurity
- · Be skeptical about claims that keeping source code secret makes the system significantly more secure

```
Joseph CS161 ©UCB Fall 2006
```

10/18/06

Lec 14, 17



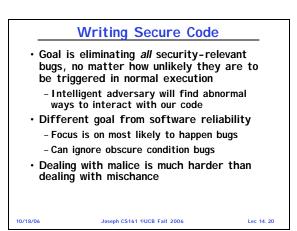
### **Administrivia**

- Grading policy
  - We use EECS upper division class guidelines » Overall class GPA 2.7 - 3.1, avg grade B or B+
  - Roughly 23% A's, 50% B's, 20% C's, 5% D's, and 2% F's
- · Midterm grade reports for potential D's and F's have been posted to BearFacts
  - If you receive a notice, see your TA or one of the profs
  - If you skipped HW#1, don't skip others
- · Projects will have a journal details in section

10/18/06

Joseph CS161 ©UCB Fall 2006

Lec 14, 19



(1) Modularity and decompo security	sition for
(2) Formal reasoning about invariants	code using
(3) Defensive programming	
In the next lecture, we'll di programming language-speci integrating security into the lifecycle	fic issues and

· Decompose well-designed system into modules - All interactions through well-defined interfaces

Modularity

- Each module performs a clear function » "What functionality it provides" not "how it is implemented"
- · Granularity depends on system and language
  - A module typically has state and code
  - In Java (object-oriented), a class (or a few closely related classes)
  - In C, its own file with a clear external interface, along with many internal functions that are not externally visible or callable

10/18/06

Joseph CS161 ©UCB Fall 2006

### Lec 14.22

### Module Design

- Focus on interface design
  - Interface is the caller-callee contract
  - Should change less often than implementation
  - Caller only needs to understand interface
  - Should interact only through defined interface » No global variables for communication
- · A module is a blob
  - The interface is its surface area
  - The implementation is its volume
- Thoughtful design has narrow and conceptually clean interfaces and modules have low surface area to volume ratio 10/18/0 Lec 14.23

```
Joseph CS161 ©UCB Fall 2006
```

Module	Decomposition Sugg	estions
Module	Decomposition Sugg	CSTIONS
- Conta (buffe	e the harm caused by mod in damage from module pene er overrun) or unexpected b mentation bug)	tration
<ul> <li>Draw a module</li> </ul>	security perimeter around	l each
	one misbehaving module fror modules' behaviors	n changing
• Plan for	r failure:	
	in advance about consequen e being compromised	ces of each
- Struc	ture system to reduce conse	equences
0/18/06	Joseph CS161 ©UCB Fall 2006	Lec 14. 24

### Monolithic Architecture

- · All modules in a common address space
  - Unecessary security risk: compromise one module and all others can be penetrated
- Alternatives:

10/18/0

- Java isolates modules using type-safety
- Languages like C require placing each module in its own process to protect it
- Follow principle of least privilege at a module granularity
  - Provide each module with the least privilege necessary to get its job done
  - Architect system so most modules need only minimal privileges

Joseph CS161 ©UCB Fall 2006

### Module Design with Least Privilege

- Can you structure a complex system of computations that require lots of code so they're isolated in modules with few privileges?
- Modules with extra privileges should have very little code
  - The more privilege for a module, the greater the confidence we need that it is correct
  - More confidence generally requires less code...

Joseph CS161 ©UCB Fall 2006

Lec 14. 26

10/18/06

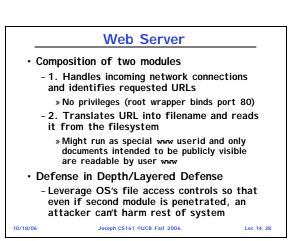
### Module Example

- Break up a network server listening on a port below 1024 into two pieces:
  - Small start-up wrapper and the app itself
  - Binding to 0 1023 port requires root privileges, so let wrapper run as root, bind to desired port, and then spawn the app passing it the bound port
- The app itself then runs as non-root user - Limits damage if app is compromised
- Wrapper can be written in a few dozen lines of code making thorough validation possible

10/18/06

Joseph CS161 ©UCB Fall 2006

Lec 14.27



### Reasoning About Code

- Functions make certain assumptions about their arguments
  - Caller must make sure assumptions are valid
  - These are often called preconditions
- Precondition for f() is an assertion (a logical proposition) that must hold at input to f()
  - Function f() must behave correctly if its preconditions are met
  - If any precondition is not met, all bets are off
- Caller must call f() such that preconditions true – an obligation on the caller, and callee may freely assume obligation has been met

10/18/06

Joseph CS161 ©UCB Fall 2006

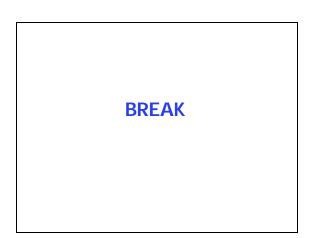
Simple Precondition Examp	le
<pre>• /* Requires: p != NULL */ int deref(int *p) {     return *p; }</pre>	
• Unsafe to dereference a null pointer	r
<ul> <li>Impose precondition that caller of d must meet: p ? NULL holds at entra deref()</li> </ul>	
<ul> <li>If all callers ensure this precondition will be safe to call deref()</li> </ul>	on, it
<ul> <li>Can combine assertions using logical connectives (and, or, implication)</li> </ul>	
<ul> <li>Also existentially and universally qua logical formulas</li> </ul>	
/18/06 Joseph CS161 ©UCB Fall 2006	Lec 14.30

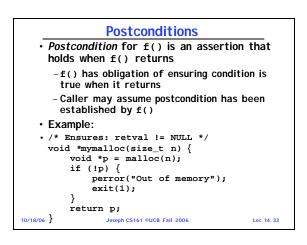


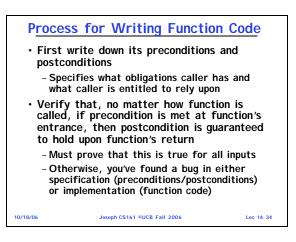
```
/* Requires:
    a != NULL
    for all j in 0..n-1, a[j] != NULL */
    int sum(int *a[], size_t n) {
        int total = 0, i;
        for (i=0; icn; i++)
            total += *(a[i]);
        return total;
    }
• Second precondition:
    -Forall j.(0 = j < n) ? a[j]?NULL
    - If you're comfortable with formal logic, write
    your assertions this way for precision
```

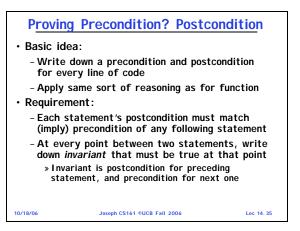
Not necessary to be so formal

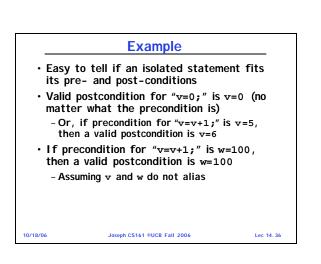
- Goal is to think explicitly about assumptions and communicate requirements to others

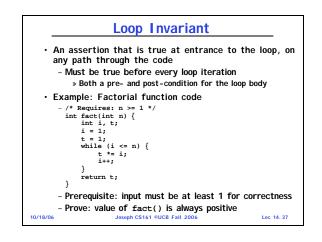


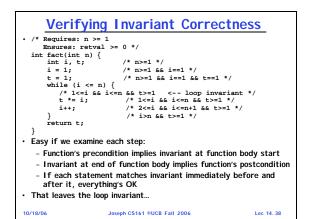










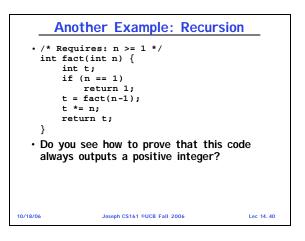


Verifying the Loop Invariant
• Loop invariant: 1<=i && i<=n && t>=1
<ul> <li>Prove it is true at start of first loop iteration</li> </ul>
- Follows from:
» n=1 Ù i=1 Ù t=1 ? 1=i=n Ù t=1
<pre>» if i=1, then certainly i=1</pre>
<ul> <li>Prove that if it holds at start of any loop iteration, then it holds at start of next iteration (if there's one)</li> </ul>
- True, since invariant at end of loop body 2=i=n+1 $\tilde{U}$ t=1 and loop termination condition i=n implies invariant at start of loop body 1=i=n $\tilde{U}$ t=1
<ul> <li>Follows by induction on number of iterations that loop invariant is always true on entrance to loop body</li> </ul>
<ul> <li>Thus, fact() will always make postcondition true, as precondition is established by its caller</li> </ul>

10/18/06

Joseph CS161 ©UCB Fall 2006

Lec 14.39



Analysis
<pre>• /* Requires: n &gt;= 1 Ensures: retval &gt;= 0 */</pre>
int fact(int n) {
int t;
if (n == 1)
return 1; /* n>=2 */
t = fact(n-1); /* t>=0 */
t *= n; /* t>=0 */
return t;
<ul> <li>Before recursive call to fact(), we know:         <ul> <li>n=1 (by precondition), n?1 (since if stmt didn't follow then branch), and n is an integer</li> </ul> </li> </ul>
<ul> <li>Follows that n=2, or n-1=1 (means precondition is met when making recursive call)</li> </ul>
• Can conclude that fact(n-1) return value is positive from postcondition for fact()
10/18/06 Joseph CS161 ©UCB Fall 2006 Lec 14. 41

		• • • •
Fun	iction Post-/Pre-Condit	ions
	ime we see a function call, w rify that its precondition will	
	en we can conclude its postcond ds and use this fact in our reas	
	tating every function with pre conditions enables <i>modular re</i>	
cod	verify function £() by looking e and the annotations on every calls	
	Can ignore code of all other functi functions called transitively	ons and
	kes reasoning about £() an alm ely local activity	ost
10/18/06	Joseph CS161 ©UCB Fall 2006	Lec 14, 42

### Documentation

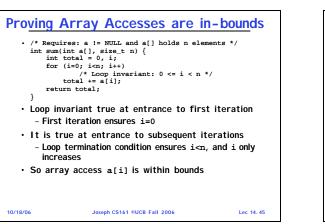
- Pre-/post-conditions serve as useful documentation
  - To invoke Bob's code, Alice only has to look at pre- and post-conditions - she doesn't need to look at or understand his code
- Useful way to coordinate activity between multiple programmers:
  - Each module assigned to one programmer, and pre-/post-conditions are a contract between caller and callee
  - Alice and Bob can negotiate the interface (and responsibilities) between their code at design time

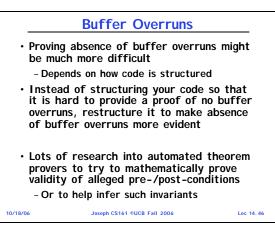
10/18/06

Joseph CS161 ©UCB Fall 2006

Lec 14, 43

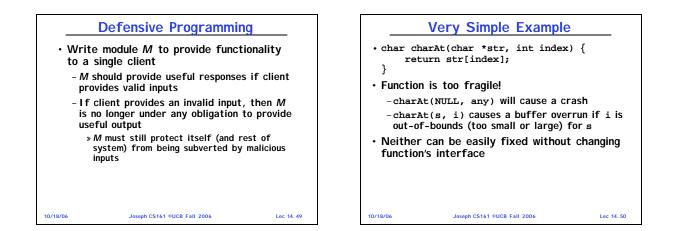
# Acoiding Security Holes Acoiding Security Holes Acoid security holes (or program crashes) Ame implicit requirements code must meet Must not divide by zero, make out-of-bounds Must not divide by zero, make out-of-bounds Anst not divide by zero, make out-of-bounds

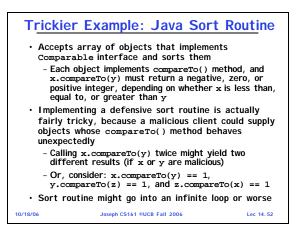




## Pre-/Post-Condition Summary Looks tedious, but gets easier over time With practice you can avoid writing down detailed invariants before every statement Think about data structures and code in terms of invariants first, then write the code Usually can avoid formal notation, omit obvious parts, and only write down important ones Usually writing down pre-/post-conditions and loop invariant for every loop is enough Reasoning about code takes time and energy Worth it for highly secure code

Defensive Programming	
Like defensive driving, but for code:	
<ul> <li>Avoid depending on others, so that if they do something unexpected, you won't crash - survive unexpected behavio</li> </ul>	
<ul> <li>Software engineering focuses on functionality:</li> </ul>	
- Given correct inputs, code produces useful/correct outputs	
<ul> <li>Security cares about what happens when program is given invalid or unexpected inputs:</li> </ul>	
<ul> <li>Shouldn't crash, cause undesirable side-effects, or produce dangerous outputs for bad inputs</li> </ul>	
• Defensive programming	
- Apply idea at every interface or security perimeter	
$\ensuremath{\scriptscriptstyle >}\xspace$ So each module remains robust even if all others misbehave	
• General strategy	
<ul> <li>Assume attacker controls module's inputs, make sure nothing terrible happens</li> </ul>	
10/18/06 Joseph CS161 ©UCB Fall 2006 Lec 14. 48	





### Some General Advice

- 1. Check for error conditions
  - Always check return values of all calls (assuming this is how they indicate errors)
  - In languages with exceptions, can locally handle it or propagate (expose) to caller
  - Check error paths very carefully
  - » Often poorly tested, so they often contain memory leaks and other bugs
- What if you detect an error condition?
  - For expected errors, try to recover
  - Harder to recover from unexpected errors
  - Always safe to abort processing and terminate if an error condition is signaled (fail-stop behavior) Joseph CS161 ©UCB Fall 2006

### 10/18/06

Lec 14, 53