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

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

3. *Use Fail-Safe Defaults*

- Use ***default-deny*** polices
  - Start by denying all access, then allow only that which has been explicitly permitted
- Ensures that if security mechanisms fail or crash, default will be secure behavior
- Example: Packet filter is a router
  - Failure means no packets will be routed
    - » Fail-safe behavior
  - Fail-open behavior much more dangerous
    - » Attacker just waits for packet filter to crash (or induces crash) and then the fort is wide open!

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

4. *Separation of Responsibility*

- Split up privilege
  - No one person or program has complete power
  - Require more than one party to approve before access is granted
- Two-party rule examples
  - 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
    - » Helps prevent insider fraud (under-/over-charge)
  - Most companies: purchases over certain amount must be approved by both requesting employee and a purchasing officer
    - » Helps prevent insider fraud in vendor choice

Nuclear Two-Party Rule

• Minuteman nucle

5. *Defense in Depth*

• A closely related principle
  – "You can recognize a security guru because they're wearing both a belt and a set of suspenders"

• Principle is that with multiple redundant protections, all of them have to be breached to endanger system security

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

7. *Usability*

• Security systems must be usable by ordinary people and take into account humans' role
• Example
  – Web browser pops up security warnings, but no indication of steps you should take
    » What do you do? Like everyone else click "OK"…
  – NSA's crypto equipment stores key material on small physical token shaped like ordinary key
    » To activate encryption device, insert key into device's slot and turn it
    » Intuitively understandable interface, even for 18-year-olds soldiers with minimal training

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

9. *Least Common Mechanism*

• Be careful with shared code!.
  – Original assumptions may no longer be valid
  – Threat model may have changed
• Example: Internet users were once only researchers, who trusted each other
  – Most networking protocols designed during those days assumed that all other network participants were benign and non-malicious
  – Not true today! Millions of users, many malicious ones…
  – Many old network protocols are suffering under the strain of attack (e.g., spam)

10. *Detect if You Can't Prevent*

- If you can't prevent break-ins, at least detect them and provide a way to identify the perpetrator
- Forensics are important
  - Keep audit logs so you can analyze break-ins afterwards
- Example: FIPS 140-1 federal standard for tamper-resistant hardware
  - Type III devices (highest level) are very expensive
  - Type II devices are only required to be tamper-evident (e.g., a visibly broken seal)
    - » Lower cost and usable in broad set of apps

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

12. *Don't Rely on Security Through Obscurity*

- We've seen this one in the last lecture…
- 'Security through obscurity' phrase
  - Systems that rely on secrecy of design, algorithms, or source code to be secure
- Claimed reasoning:
  - "This system is so obscure, only 100 people understand anything about it, so what are the odds that adversaries will bother attacking it?"
- Self-defeating approach
  - As system becomes more popular, more incentive to attack it, and cannot rely on its obscurity to keep attackers away…

Secret Designs

- Very hard to keep system design secret from a dedicated adversary
  - Every running installation has binary executable code that can be disassembled
  - Hard to assess chances that secret will leak or difficulty of learning the secret
- If secret ever leaks, can be hard to update widely-deployed systems
  - No recourse if someone ever succeeds
- History has a lousy track record
  - Many systems that have relied upon code or design secrecy for security have failed miserably

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

13. *Design Security in, From the Start*

- Often doesn't work to retrofit security into an existing implemented application
  - Stuck with chosen architecture
  - Can't change system decomposition to ensure any of the good principles we discussed
- Backwards compatibility often particularly painful, because you have to support worst insecurities of all previous versions

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

Writing Secure Code

- Goal is eliminating *all* security-relevant bugs, no matter how unlikely they are to be triggered in normal execution
  - Intelligent adversary will find abnormal ways to interact with our code
- Different goal from software reliability
  - Focus is on most likely to happen bugs
  - Can ignore obscure condition bugs
- Dealing with malice is much harder than dealing with mischance

Three Fundamental Techniques

- (1) Modularity and decomposition for security
- (2) Formal reasoning about code using invariants
- (3) Defensive programming

- In the next lecture, we'll discuss programming language-specific issues and integrating security into the software lifecycle

Modularity

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

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

Module Decomposition Suggestions

- Minimize the harm caused by module failure
  - Contain damage from module penetration (buffer overrun) or unexpected behavior (implementation bug)
- Draw a security perimeter around each module
  - Keep one misbehaving module from changing other modules' behaviors
- Plan for failure:
  - Think in advance about consequences of each module being compromised
  - Structure system to reduce consequences

Monolithic Architecture

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

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…

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

Web Server

- Composition of two modules
  - 1. Handles incoming network connections and identifies requested URLs
    - » No privileges (root wrapper binds port 80)
  - 2. Translates URL into filename and reads it from the filesystem
    - » Might run as special `www` userid and only documents intended to be publicly visible are readable by user `www`
- Defense in Depth/Layered Defense
  - Leverage OS's file access controls so that even if second module is penetrated, an attacker can't harm rest of system

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

Simple Precondition Example

-
  ```
  /* Requires: p != NULL */
  int deref(int *p) {
      return *p;
  }
  ```
- Unsafe to dereference a null pointer
  - Impose precondition that caller of `deref()` must meet: *p ? NULL* holds at entrance to `deref()`
- If all callers ensure this precondition, it will be safe to call `deref()`
- Can combine assertions using logical connectives (and, or, implication)
  - Also existentially and universally quantified logical formulas

Another Example

-
  ```
  /* 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; i<n; 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

BREAK

Postconditions

- *Postcondition* for `f()` is an assertion that holds when `f()` returns
  - `f()` has obligation of ensuring condition is true when it returns
  - Caller may assume postcondition has been established by `f()`
- Example:

-
  ```
  /* Ensures: retval != NULL */
  void *mymalloc(size_t n) {
      void *p = malloc(n);
      if (!p) {
          perror("Out of memory");
          exit(1);
      }
      return p;
  }
  ```

Process for Writing Function Code

- First write down its preconditions and postconditions
  - Specifies what obligations caller has and what caller is entitled to rely upon
- Verify that, no matter how function is called, if precondition is met at function's entrance, then postcondition is guaranteed to hold upon function's return
  - Must prove that this is true for all inputs
  - Otherwise, you've found a bug in either specification (preconditions/postconditions) or implementation (function code)

Proving Precondition? Postcondition

- Basic idea:
  - Write down a precondition and postcondition for every line of code
  - Apply same sort of reasoning as for function
- Requirement:
  - Each statement's postcondition must match (imply) precondition of any following statement
  - At every point between two statements, write down *invariant* that must be true at that point
    - » Invariant is postcondition for preceding statement, and precondition for next one

Example

- Easy to tell if an isolated statement fits its pre- and post-conditions
- Valid postcondition for "`v=0;`" is `v=0` (no matter what the precondition is)
  - Or, if precondition for "`v=v+1;`" is `v=5`, then a valid postcondition is `v=6`
- If precondition for "`v=v+1;`" is `w=100`, then a valid postcondition is `w=100`
  - Assuming `v` and `w` do not alias

Loop Invariant

- An assertion that is true at entrance to the loop, on any path through the code
  - Must be true before every loop iteration
    - » Both a pre- and post-condition for the loop body
- Example: Factorial function code

```
– /* Requires: n >= 1 */
  int fact(int n) {
      int i, t;
      i = 1;
      t = 1;
      while (i <= n) {
          t *= i;
          i++;
      }
      return t;
  }
```

  - Prerequisite: input must be at least 1 for correctness
  - Prove: value of `fact()` is always positive

Verifying Invariant Correctness

```
• /* Requires: n >= 1
     Ensures: retval >= 0 */
  int fact(int n) {
      int i, t;           /* n>=1 */
      i = 1;              /* n>=1 && i==1 */
      t = 1;              /* n>=1 && i==1 && t==1 */
      while (i <= n) {
        /* 1<=i && i<=n && t>=1   <-- loop invariant */
        t *= i;           /* 1<=i && i<=n && t>=1 */
        i++;              /* 2<=i && i<=n+1 && t>=1 */
      }                   /* i>n && t>=1 */
      return t;
  }
```

- Easy if we examine each step:
  - Function's precondition implies invariant at function body start
  - Invariant at end of function body implies function's postcondition
  - If each statement matches invariant immediately before and after it, everything's OK
- That leaves the loop invariant…

Verifying the Loop Invariant

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

Another Example: Recursion

```
• /* Requires: n >= 1 */
  int fact(int n) {
      int t;
      if (n == 1)
          return 1;
      t = fact(n-1);
      t *= n;
      return t;
  }
```

- Do you see how to prove that this code always outputs a positive integer?

Analysis

```
• /* Requires: n >= 1
     Ensures: retval >= 0 */
  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;
  }
```

- Before recursive call to `fact()`, we know:
  - `n=1` (by precondition), `n?1` (since if stmt didn't follow then branch), and `n` is an integer
  - Follows that `n=2`, or `n-1=1` (means precondition is met when making recursive call)
- Can conclude that `fact(n-1)` return value is positive from postcondition for `fact()`

Function Post-/Pre-Conditions

- Any time we see a function call, we have to verify that its precondition will be met
  - Then we can conclude its postcondition holds and use this fact in our reasoning
- Annotating every function with pre- and post-conditions enables *modular reasoning*
  - Can verify function `f()` by looking only its code and the annotations on every function `f()` calls
    - » Can ignore code of all other functions and functions called transitively
  - Makes reasoning about `f()` an almost purely local activity

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

Avoiding Security Holes

- To avoid security holes (or program crashes)
  - Some implicit requirements code must meet
    - » Must not divide by zero, make out-of-bounds memory accesses, or deference null ptrs, …
- We can try to prove that code meets these requirements using same style of reasoning
  - Ex: when a pointer is dereferenced, there is an implicit precondition that pointer is non-null and in-bounds

Proving Array Accesses are in-bounds

```
/* Requires: a != NULL and a[] holds n elements */
int sum(int a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
            /* Loop invariant: 0 <= i < n */
        total += a[i];
    return total;
}
```

- Loop invariant true at entrance to first iteration
  - First iteration ensures `i=0`
- It is true at entrance to subsequent iterations
  - Loop termination condition ensures `i<n`, and `i` only increases
- So array access `a[i]` is within bounds

Buffer Overruns

- Proving absence of buffer overruns might be much more difficult
  - Depends on how code is structured
- Instead of structuring your code so that it is hard to provide a proof of no buffer overruns, restructure it to make absence of buffer overruns more evident
- Lots of research into automated theorem provers to try to mathematically prove validity of alleged pre-/post-conditions
  - Or to help infer such invariants

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:
  - Avoid depending on others, so that if they do something unexpected, you won't crash – survive unexpected behavior
- Software engineering focuses on functionality:
  - Given correct inputs, code produces useful/correct outputs
- Security cares about what happens when program is given invalid or unexpected inputs:
  - Shouldn't crash, cause undesirable side-effects, or produce dangerous outputs for bad inputs
- Defensive programming
  - Apply idea at every interface or security perimeter
    - » So each module remains robust even if all others misbehave
- General strategy
  - Assume attacker controls module's inputs, make sure nothing terrible happens

Defensive Programming

- Write module *M* to provide functionality to a single client
  - *M* should provide useful responses if client provides valid inputs
  - If client provides an invalid input, then *M* is no longer under any obligation to provide useful output
    - » *M* must still protect itself (and rest of system) from being subverted by malicious inputs

Very Simple Example

```
• char charAt(char *str, int index) {
      return str[index];
  }
```

- Function is too fragile!
  - `charAt(NULL, any)` will cause a crash
  - `charAt(s, i)` causes a buffer overrun if `i` is out-of-bounds (too small or large) for `s`
- Neither can be easily fixed without changing function's interface

Another Simple Example with Many Flaws

```
• char *double(char *str) {
      size_t len = strlen(str);
      char *p = malloc(2*len+1);
      strcpy(p, str);
      strcpy(p+len, str);
      return p;
  }
```

- `double(NULL)` will cause a crash
  - Fix: test if `str` is a null ptr, and if so, return `NULL`
- Return value of `malloc()` is not checked
  - If out-of-memory, `malloc()` will return null ptr and call to `strcpy()` will cause program crash
  - Fix: test return value of `malloc()`
- If `str` is very long, then expression `2*len+1` will overflow, potentially causing a buffer overrun
  - 2^{31} byte input str on 32-bit machine will have 1 byte allocated, and `strcpy` will immediately trigger a heap overrun

Trickier Example: Java Sort Routine

- Accepts array of objects that implements `Comparable` interface and sorts them
  - Each object implements `compareTo()` method, and `x.compareTo(y)` must return a negative, zero, or positive integer, depending on whether `x` is less than, equal to, or greater than `y`
- Implementing a defensive sort routine is actually fairly tricky, because a malicious client could supply objects whose `compareTo()` method behaves unexpectedly
  - Calling `x.compareTo(y)` twice might yield two different results (if `x` or `y` are malicious)
  - Or, consider: `x.compareTo(y) == 1`, `y.compareTo(z) == 1`, and `z.compareTo(x) == 1`
- Sort routine might go into an infinite loop or worse

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)