Secure Programming, Continued
CS 136
Computer Security
Peter Reiher
March 11, 2008

Outline

- Introduction
- Principles for secure software
- Choosing technologies
- Major problem areas
- Evaluating program security

Race Conditions

- Another common cause of security bugs
- Usually involve multiprogramming or multithreaded programs
- Caused by different threads of control operating in unpredictable fashion
  - When programmer thought they'd work in a particular order

What Is a Race Condition?

- A situation in which two (or more) threads of control are cooperating or sharing something
- If their events happen in one order, one thing happens
- If their events happen in another order, something else happens
- Often the results are unforeseen

Security Implications of Race Conditions

- Usually you checked privileges at one point
- You thought the next lines of code would run next
  - So privileges still apply
- But multiprogramming allows things to happen in between

The TOCTOU Issue

- Time of Check to Time of Use
- Have security conditions changed between when you checked?
- And when you used it?
- Multiprogramming issues can make that happen
- Sometimes under attacker control

An Example

- Code from Unix involving a temporary file

```
res = access(“/tmp/userfile”, R_OK);
If (res != 0)
  die(“access”);
fd = open(“/tmp/userfile,O_RDONLY);
```

A Short Detour

- In Unix, processes can have two associated user IDs
  – Effective ID
  – Real ID
- Real ID is the ID of the user who actually ran it
- Effective ID is current ID for access control purposes
- Setuid programs run this way
- System calls allow you to manipulate it

Effective UID and Access Permissions

- Unix checks accesses against effective UID, not real UID
- So setuid program uses permissions for the program’s owner
  - Unless relinquished
- Remember, root has universal access privileges

What's (Supposed to Be) Going on Here?

- Code ran as setuid root
- Checked access on `/tmp/userfile` to make sure user was allowed to read it
  - User can use links to control what this file is
- `access()` checks real user ID, not effective one
  - So checks access permissions not as root, but as actual user
- So if user can read it, open file for read
  - Which root is definitely allowed to do
- Otherwise exit

What's Really Going On Here?

- This program might not run uninterrupted
- OS might schedule something else in the middle
- In particular, between those two lines of code

How the Attack Works

- Attacker puts innocuous file in `/tmp/userfile`
- Calls the program
- Quickly deletes file and replaces it with link to secret file
  - One only readable by root
- If timing works, he gets secret contents

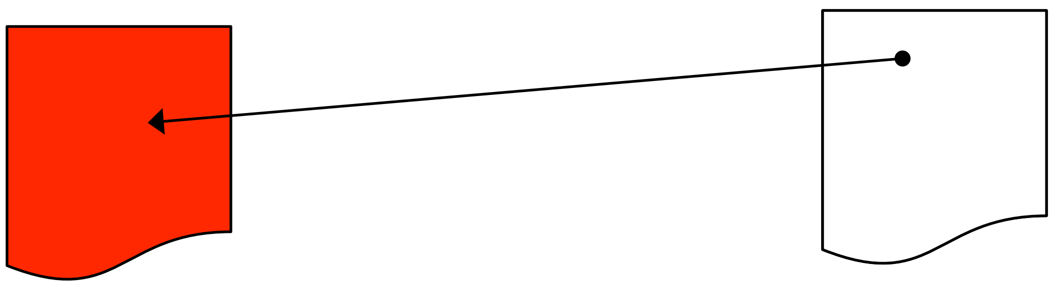
The Dynamics of the Attack

Success!

Let's try that again!

One more time!

/etc/secretfile /tmp/userfile



1. Run program
2. Change file

```
res = access("/tmp/userfile", R_OK);
if (res != 0)
      die("access");
fd = open("/tmp/userfile,O_RDONLY);
```

How Likely Was That?

- Not very
  - The timing had to be just right
- But the attacker can try it many times
  - And may be able to influence system to make it more likely
- And he only needs to get it right once
- Timing attacks of this kind can work
- The longer between check and use, the more dangerous

Some Types of Race Conditions

- File races
  - Which file you access gets changed
- Permissions races
  - File permissions are changed
- Ownership races
  - Who owns a file changes
- Directory races
  - Directory hierarchy structure changes

Preventing Race Conditions

- Minimize time between security checks and when action is taken
- Be especially careful with files that users can change
- Use locking and features that prevent interruption, when possible
- Avoid designs that require actions where races can occur

Randomness and Determinism

- Many pieces of code require some randomness in behavior
- Where do they get it?
- As key discussion showed, it's not that easy to get

Pseudorandom Number Generators

- PRNG
- Mathematical methods designed to produce strings of random-like numbers
- Actually deterministic
  - But share many properties with true random streams of numbers

Attacks on PRNGs

- Cryptographic attacks
  - Observe stream of numbers and try to deduce the function
- State attacks
  - Attackers gain knowledge of or influence the internal state of the PRNG

An Example

- ASF Software's Texas Hold'Em Poker
- Flaw in PRNG allowed cheater to determine everyone's cards
  - Flaw in card shuffling algorithm
  - Seeded with a clock value that can be easily obtained

Another Example

- Netscape’s early SSL implementation
- Another guessable seed problem
  - Based on knowing time of day, process ID, and parent process ID
  - Process IDs readily available by other processes on same box
- Broke keys in 30 seconds

How to Do Better?

- Use hardware randomness, where available
- Use high quality PRNGs
  - Preferably based on entropy collection methods
- Don't use seed values obtainable outside the program

Proper Use of Cryptography

- Never write your own crypto functions if you have any choice
- Never, ever, design your own encryption algorithm
  - Unless that's your area of expertise
- Generally, rely on tried and true stuff
  - Both algorithms and implementations

Proper Use of Crypto

- Even with good crypto algorithms (and code), problems are possible
- Proper use of crypto is quite subtle
- Bugs possible in:
  - Choice of keys
  - Key management
  - Application of cryptographic ops

An Example

- Microsoft's PPTP system
  - A planned competitor for IPSec
- Subjected to careful analysis by Schneier and Mudge
- With disappointing results
- Bugs in the implementation, not the standard

Bugs in PPTP Implementation

- Password hashing
  - Weak algorithms allow eavesdroppers to learn the user's password
- Challenge/reply authentication protocol
  - A design flaw allows an attacker to masquerade as the server
- Encryption bugs
  - Implementation mistakes allow encrypted data to be recovered

More PPTP Bugs

- Encryption key choice
  - Common passwords yield breakable keys, even for 128-bit encryption
- Control channel problems
  - Unauthenticated messages let attackers crash PPTP servers
- Don’t treat this case with contempt just because it’s Microsoft
  - They hire good programmers
  - Who nonetheless screwed up

Another Example

- An application where RSA was used to distribute a triple-DES key
- Seemed to work fine
- Someone noticed that part of the RSA key exchange were always the same
  – That's odd . . .

What Was Happening?

- Bad parameters were handed to the RSA encryption code
- It failed and returned an error
- Which wasn't checked for
  - Since it "couldn't fail"
- As a result, RSA encryption wasn't applied at all
- The session key was sent in plaintext . . .

Trust Management and Input Validation

- Don’t trust anything you don’t need to
- Don’t trust other programs
- Don’t trust other components of your program
- Don’t trust users
- Don’t trust the data users provide you

Trust

- Some trust required to get most jobs done
- But determine how much you must trust the other
  - Don't trust things you can independently verify
- Limit the scope of your trust
  - Compartmentalization helps
- Be careful who you trust

An Example of Misplaced Trust

- A Unix system from 1990s
- Supposed to only be used for email
- Menu-driven system
  - From which you selected the mailer
- But the mailer allowed you to edit messages
  - Via vi
- And vi allowed you to fork a shell
- So anyone could run any command

What Was the Trust Problem?

- The menu system trusted the mail program
  - Not to do anything but handle mail
- The mail program trusted vi
  - To do proper editing
  - Probably unaware of menu system's expectations
- vi did more
  - It wasn't evil, but it wasn't doing what was expected
-

Validating Input

- Never assume users followed any rules in providing you input
- They can provide you with anything
- Unless you check it, assume they've given you garbage
  - Or worse
- Just because the last input was good doesn't mean the next one will be

Treat Input as Hostile

- If it comes from outside your control and reasonable area of trust
- Probably even if it doesn’t
- There may be code paths you haven’t considered
- New code paths might be added
- Input might come from new sources

For Example

- Shopping cart exploits
- Web shopping carts sometimes handled as a cookie delivered to the user
- Some of these weren't encrypted
- So users could alter them
- The shopping cart cookie included the price of the goods . . .

What Was the Problem?

- The system trusted the shopping cart cookie when it was returned
  – When there was no reason to trust it
- Either encrypt the cookie
  – Making the input more trusted
  – Can you see any problem with this approach?
- Or scan the input before taking action on it
  – To find refrigerators being sold for 3 cents

General Issues of Untrusted Inputs

- Check all inputs to be sure they are what you expect
  – Format
  – Range of values
  – Matching previous history
- Especially important for inputs coming straight from the user
  – Extra especially if over the network

Evaluating Program Security

- What if your problem isn't writing secure code?
- It's determining if someone else's code is secure?
  - Or, perhaps, their overall system
- How do you go about evaluating code for security?
- Much of this material from "The Art of Software Security Assessment," Dowd, McDonald, and Schuh

Stages of Review

- You can review a program's security at different stages in its life cycle
  - During design
  - Upon completion of the coding
  - When the program is in place and operational
- Different issues arise in each case

Design Reviews

- Done perhaps before there's any code
- Just a design
- Clearly won't discover coding bugs
- Clearly could discover fundamental flaws
- Also useful for prioritizing attention during later code review

Purpose of Design Review

- To identify security weaknesses in a planned software system
- Essentially, identifying threats to the system
- Performed by a process called *threat modeling*
- Usually (but not always) performed before system is built

Threat Modeling

- Done in various ways
- One way uses a five step process:
  1. Information collection
  2. Application architecture modeling
  3. Threat identification
  4. Documentation of findings
  5. Prioritizing the subsequent implementation review

1. Information Collection

- Collect all available information on design
- Try to identify:
  - Assets
  - Entry points
  - External entities
  - External trust levels
  - Major components
  - Use scenarios

Sources of Information

- Documentation
- Interviewing developers
- Standards documentation
- Source profiling
  - If source already exists
- System profiling
  - If a working version is available

2. Application Architecture Modeling

- Using information gathered, develop understanding of the proposed architecture
- To identify design concerns
- And to prioritize later efforts
- Useful to document findings using some type of model

Modeling Tools for Design Review

- Markup languages (e.g., UML)
  – Particularly diagramming features
  – Used to describe OO classes and their interactions
  – Also components and uses
- Data flow diagrams
  – Used to describe where data goes and what happens to it

3. Threat Identification

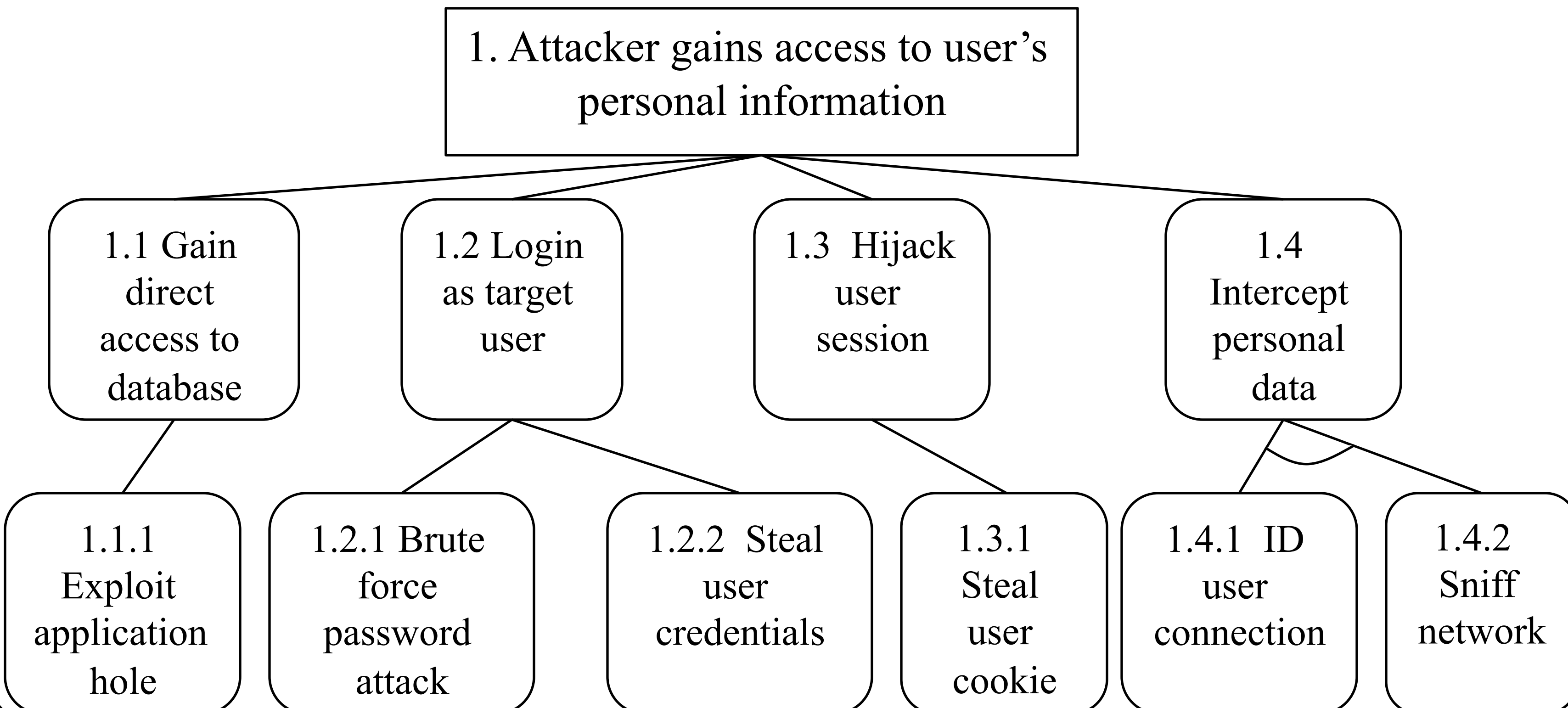
- Based on models and other information gathered
- Identify major security threats to the system's assets
- Typically done with *attack trees*

Attack Trees

- A way to codify and formalize possible attacks on a system
- Makes it easier to understand relative levels of threats
  - In terms of possible harm
  - And probability of occurring

A Sample Attack Tree

- For a web application



4. Documentation of Findings

- Summarize threats found
  - Give recommendations on addressing each
- Generally best to prioritize threats
  - How do you determine priorities?

DREAD Risk Ratings

- Assign number from 1-10 on these categories:
- **D**amage potential
- **R**eproducibility
- **E**xploitability
- **A**ffected users
- **D**iscoverability
- Gives better picture of important issues for each threat

5. Prioritizing Implementation Review

- Review of actual implementation should follow review of design
- Immediately, if implementation already available
- Later, if implementation not mature yet
- Need to determine how to focus your efforts in this review

Why Prioritize?

- There are usually many threats
- Implementation reviews require a lot of resources
- So you probably can't look very closely at everything
- Need to decide where to focus limited amount of attention

One Prioritization Approach

- Make a list of the major components
- Identify which component each risk (identified earlier) belongs to
- Total the risk scores for categories
- Use the resulting numbers to prioritize

Application Review

- Reviewing a mature (possibly complete) application
- A daunting task if the system is large
- And often you know little about it
  - Maybe you performed a design review
  - Maybe you read design review docs
  - Maybe less than that
- How do you get started?

Need to Define a Process

- Don't just dive into the code
- Process should be:
  - Pragmatic
  - Flexible
  - Results oriented
- Will require code review
  - Which is a skill one must develop

Review Process Outline

1. Preassessment
   – Get high level view of system
2. Application review
   – Design review, code review, maybe live testing
3. Documentation and analysis
4. Remediation support
   – Help them fix the problems

Reviewing the Application

- You start off knowing little about the code
- You end up knowing a lot more
- You'll probably find the deepest problems related to logic after you understand things
- A design review gets you deeper quicker
  - So worth doing, if not already done
- The application review will be an iterative process

General Approaches To Design Reviews

- Top-down
  - Start with high level knowledge, gradually go deeper
- Bottom-up
  - Look at code details first, build model of overall system as you go
- Hybrid
  - Switch back and forth, as useful

Code Auditing Strategies

- Code comprehension (CC) strategies
  - Analyze source code to find vulnerabilities and increase understanding
- Candidate point (CP) strategies
  - Create list of potential issues and look for them in code
- Design generalization (DG) strategies
  - Flexibly build model of design to look for high and medium level flaws

Some Example Strategies

- Trace malicious input (CC)
  - Trace paths of data/control from points where attackers can inject bad stuff
- Analyze a module (CC)
  - Choose one module and understand it
- Simple lexical candidate points (CP)
  - Look for text patterns (e.g., `strcpy()`)
- Design conformity check (DG)
  - Determine how well code matches design

Guidelines for Auditing Code

- Perform flow analysis carefully within functions you examine
- Re-read code you've examined
- Desk check important algorithms
- Use test cases for important algorithms
  - Using real system or desk checking
  - Choosing inputs carefully

Useful Auditing Tools

- Source code navigators
- Debuggers
- Binary navigation tools
- Fuzz-testing tools
  - Automates testing of range of important values

Conclusion

- Many computer security problems are rooted in insecure programming
- We have scratched the surface of the topic here
- Similarly, we've scratched the surface of auditing issues
- If your job is coding or auditing, you'll need to dig deeper yourself