Secure Programming
CS 136
Computer Security
Peter Reiher
March 6, 2008

Outline

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

Introduction

- How do you write secure software?
- Basically, define security goals
- And use techniques that are likely to achieve them
- Ideally, part of the whole process of software development
  - Not just some tricks programmers use

Designing for Security

- Often developers design for functionality
  - “We’ll add security later”
- Security retrofits have a terrible reputation
  - Insecure designs offer too many attack opportunities
- Designing security from the beginning works better

For Example,

- Windows 95 and its relatives
- Not designed with security in mind
- Security professionals assume any networked Windows 95 machine can be hacked
  - Despite later security retrofits

Defining Security Goals

- Think about which security properties are relevant to your software
  - Does it need limited access?
  - Privacy issues?
  - Is availability important?
- And the way it interacts with your environment
  - Even if it doesn't care about security, what about the system it runs on?

Some Common Kinds of Problems

- We've seen these before:
  - Eavesdropping
  - Tampering
  - Spoofing and replay
  - Allowing improper access
  - Social engineering
- Many threats are *malicious input problems*

Security and Other Goals

- Security is never the only goal of a piece of software
- Usually not the primary goal
- Generally, secure software that doesn’t meet its other goals is a failure
- Consider the degree of security required as an issue of *risk*

Managing Software Security Risk

- How much risk can this software tolerate?
- What compromises can you make to minimize that risk?
  - Often other goals conflict with security
  - E.g., should my program be more usable or require strong authentication?
- Considering tradeoffs in terms of risks can clarify what you need to do

Risk Management and Software Development

- Should consider security risk as part of your software development model
- E.g., in spiral model, add risk analysis phase to the area of spiral where you evaluate alternatives
- Considering security and risks early can avoid pitfalls later
- Returning to risk when refining is necessary

Design and Security Experts

- Someone on a software development team should understand security
  - The more they understand it, the better
  - Ideally, someone on team should have explicit security responsibility
- Experts should be involved in all phases
  - Starting from design

Principles for Secure Software

- Following these doesn't guarantee security
- But they touch on the most commonly seen security problems
- Thinking about them is likely to lead to more secure code

1. Secure the Weakest Link

- Don’t consider only a single possible attack
- Look at all possible attacks you can think of
- Concentrate most attention on most vulnerable elements

For Example,

- Attackers are not likely to break cryptography
  - Switching from DES to AES probably doesn't address your weakest link
- More likely to use a buffer overflow to break in
  - And read data before it's encrypted
  - Spend the time on preventing that

2. Practice Defense in Depth

- Try to avoid designing software so failure anywhere compromises everything
- Also try to protect data and applications from failures elsewhere in the system
- Don’t let one security breach give away everything

For Example,

- Protecting data moving between servers in a single enterprise system
- Don’t just put up a firewall around whole system
- Also encrypt data in transit
- And put another firewall on each machine/application

3. Fail Securely

- Common source of security problems arise when programs fail
- Often fail into modes that aren't secure
- So attackers cause them to fail
  - And see if that helps them
- So make sure that when ordinary measures fail, the backup is secure

For Example,

- A major security flaw in typical Java RMI implementations
- If server wants to use security protocol client doesn't have, what happens?
  - Client downloads it from the server
  - Which is doesn't trust yet . . .
- Malicious entity can force installation of compromised protocol

4. Use Principle of Least Privilege

- Give minimum access necessary
- For the minimum amount of time required
- Always possible that the privileges you give will be abused
  - Either directly or through finding a security flaw
- The less you give, the lower the risk

For Example,

- In traditional Unix systems, can’t bind to port number < 1024 unless you’re root
- So if someone legitimately needs to bind to such a port, must give them root
- But once they’ve bound to it, program should relinquish privileges
- So only program flaws in limited part of program give attacker root privilege

5. Compartmentalize

- Divide programs into pieces
- Ensure that compromise of one piece does not automatically compromise others
- Set up limited interfaces between pieces
  – Allowing only necessary interactions

For Example,

- Traditional Unix has terrible compartmentalization
  - Obtaining root privileges gives away the entire system
- Redesigns that allow previous root programs to run under other identities helps
  - E.g., mail server and print server users

6. Value Simplicity

- Complexity is the enemy of security
- Complex systems give more opportunities to screw up
- Also, harder to understand all "proper" behaviors of complex systems
- So favor simple designs over complex ones

For Example,

- Re-use components when you think they're secure
- Use one implementation of encryption, not several
  – Especially if you use "tried and true" implementation
  – And one that only does what you need
  – Implementation of exactly what you need better than "Swiss army knife"

Especially Important When Human Users Involved

- Users will not read documentation
  - So don't rely on designs that require them to
- Users are lazy
  - They'll ignore pop-ups and warnings
  - "Given the choice between dancing pigs and security, users will pick dancing pigs every time." Ed Felten

7. Promote Privacy

- Avoid doing things that will compromise user privacy
- Don’t ask for data you don’t need
- Avoid storing user data permanently
  - Especially unencrypted data
- There are strong legal issues related to this, nowadays

For Example,

- Storing user passwords
- If you store them in plaintext, you can “remind” users who forget
- But breakins might compromise all of them
  – And users might use them elsewhere
- Consider storing them only encrypted
  – Which has usability issues . . .

8. Remember That Hiding Secrets is Hard

- Assume anyone who has your program can learn <u>everything</u> about it
- "Hidden" keys and passwords in executables are invariably found
- Security based on obfuscated code is always broken
- Just because you're not smart enough to crack it doesn't mean the hacker isn't, either

For Example,

- Digital rights management software often needs to hide a key
- But needs that key available to the users
- All schemes developed to do this have been cracked
  - Nowadays, usually cracked before official release of "protected" media

9. Be Reluctant to Trust

- Don’t automatically trust things
  - Especially if you don’t have to
- Remember, you’re not just trusted the honesty of the other party
  - You’re also trusting their caution
- Avoid trusting users you don’t need to trust, too
  - Doing so makes you more open to social engineering attacks

For Example,

- Why do you trust that shrinkwrapped software?
- Or that open source library?
- Must you?
- Can you design the system so it’s secure even if that component fails?
- If so, do it

10. Use Your Community Resources

- Favor widely used and respected security software over untested stuff
  - Especially your own . . .
- Keep up to date on what's going on
  - Not just patching
  - Also things like attack trends

For Example,

- Don’t implement your own AES code
- Rely on one of the widely used versions
- But also don’t be too trusting
  – E.g., just because it’s open source doesn’t mean it’s more secure

Choosing Technologies

- Different technologies have different security properties
  - Operating systems
  - Languages
  - Object management systems
  - Libraries
- Important to choose wisely
  - And understand the implications of the choice

Choices and Practicalities

- You usually don’t get to choose the OS
- The environment you’re writing for dictates the choice
  - E.g., commercial software often must be written for Windows
  - Or Linux is the platform in your company
- Might not get choice in other areas, either
  - But exercise it when you can

Operating System Choices

- Rarely an option
- If they are, does it matter?
- Probably not, any more
  - All major choices have poor security histories
    - No, Linux is not necessarily safer than Windows
  - All have exhibited lots of problems
  - In many cases, problems are in the apps, anyway
- Exception if you get to choose really trusted platform
  - E.g., SE Linux or Trusted Solaris
    - Not perfect, but better
    - But at a cost

Language Choices

- More likely to be possible
  - Though often hard to switch from what's already being used
- If you do get the choice, what should it be?

C and C++

- Probably the worst security choice
- Far more susceptible to buffer overflows than other choices
- Also prone to other reliability problems
- Often chosen for efficiency
  - But is efficiency that important for <u>your</u> application?

Java

- Less susceptible to buffer overflows
- Also better error handling than C/C++
- Has special built-in security features
  – Which aren't widely used
- But has its own set of problems
- E.g., exception handling issues
- 19 serious security flaws found between 1996 and 2001

Scripting Languages

- Depends on language
- Javascript and CGIbin have awful security reputations
- Perl offers some useful security features
- But there are some general issues

General Security Issues for Scripting Languages

- Might be security flaws in their interpreters
  - More likely than in compilers
- Scripts often easily examined by attackers
  - Obscurity of binary no guarantee, but it is an obstacle
- Scripting languages often used to make system calls
  - Inherently dangerous

Other Choice Issues

- Which distributed object management system?
  - CORBA, DCOM, RMI, .net?
  - Each has different security properties
- Which existing components to include?
- Which authentication technology to use?

Open Source vs. Closed Source

- Some argue open source software is inherently more secure
- The "more eyes" argument –
  - Since anyone can look at open source code,
  - More people will examine it
  - Finding more bugs
  - Increasing security

Is the "Many Eyes" Argument Correct?

- Probably not
- At least not in general
- Linux has security bug history similar to Windows
- Other open source projects even worse
  - In many cases, nobody really looks at the code
  - Which is no better than closed source

The Flip Side Argument

- "Hackers can examine open source software and find its flaws"
- Well, Windows' security history is not a recommendation for this view
- Most commonly exploited flaws can be found via black-box approach
  – E.g., typical buffer overflows

The Upshot?

- No solid evidence that open source or closed source produces better security
- Major exception is crypto
  - At least for crypto standards
  - Maybe widely used crypto packages
  - Criticality and limited scope means many eyeballs will really look at it

Major Security Issues for Secure Design and Coding

- Buffer overflows
- Access control issues
- Race conditions
- Randomness and determinism
- Proper use of cryptography
- Trust management and input validation

Buffer Overflows

- The poster child of insecure programming
- One of the most commonly exploited types of programming error
- Technical details of how they occur discussed earlier
- Key problem is language does not check bounds of variables

Preventing Buffer Overflows

- Use a language with bounds checking
  - Most modern languages other than C and C++
  - Not always a choice
  - Or the right choice
- Check bounds carefully yourself
- Avoid constructs that often cause trouble

Problematic Constructs for Buffer Overflows

- Most frequently C system calls:
  - `gets()`, `strcpy()`, `strcat()`, `sprintf()`, `scanf()`, `sscanf()`, `fscanf()`, `vfscanf()`, `vsprintf()`, `vscanf()`, `vsscanf()`, `streadd()`, `strecpy()`
  - There are others that are also risky

Why Are These Calls Risky?

- They copy data into a buffer
- Without checking if the length of the data copied is greater than the buffer
- Allowing overflow of that buffer
- Assumes attacker can put his own data into the buffer
  - Not always true
  - But why take the risk?

What Do You Do Instead?

- Many of the calls have variants that specify how much data is copied
  - If used properly, won't allow the buffer to overflow
- Those without the variants allow precision specifiers
  - Which limit the amount of data handled

Is That All I Have To Do?

- No
- These are automated buffer overflows
- You can easily write your own
- Must carefully check the amount of data you copy if you do
- And beware of integer overflow problems

An Example

- Actual bug in OpenSSH server:

```
u_int nresp;
. . .
nresp = packet_get_int();
If (nresp > 0) {
  response = xmalloc(nresp * sizeof(char *));
  for (i=0; i<nresp;i++)
      response[i] = packet_get_string(NULL);
}
packet_check_eom();
```

Why Is This a Problem?

- nresp is provided by the user
  - `nresp = packet_get_int();`
- But we allocate a buffer of nresp entries, right?
  - `response = xmalloc(nresp * sizeof(char *));`
- So how can that buffer overflow?
- Due to integer overflow

How Does That Work?

- The argument to `xmalloc()` is an unsigned int
- Its maximum value is 2^{32}-1
  - 4,294,967,295
- `sizeof(char *)` is 4
- What if the user sets `nresp` to 0x40000020?
- Multiplication is modulo 2^{32} . . .
  - So 4 * 0x40000020 is 0x80

What Is the Result?

- There are 128 entries in `response[]`
- And the loop iterates hundreds of millions of times
  – Copying data into the "proper place" in the buffer each time
- A massive buffer overflow

Other Programming Tools for Buffer Overflow Prevention

- Software scanning tools that look for buffer overflows
  – Of varying sophistication
- Use C compiler that includes bounds checking
  – Typically as an option
- Use integrity-checking programs
  – Stackguard, Rational's Purity, etc.

Access Control Issues

- Programs usually run under their user's identity
  - With his privileges
- Some programs get expanded privileges
  - Setuid programs in Unix, e.g.
- Poor programming here can give too much access

An Example Problem

- A program that runs setuid and allows a shell to be forked
  - Giving the caller a root environment in which to run arbitrary commands
- Buffer overflows in privileged programs usually give privileged access

A Real World Example

- `/sbin/dump` from NetBSD
- Ran setgid as group `tty`
  - To notify sysadmins of important events
  - Never dropped this privilege
- Result: `dump` would start program of user's choice as user `tty`
  - Allowing them to interact with other user's terminals

What To Do About This?

- Avoid running programs setuid
- If you must, don't make them root-owned
- Change back to the real caller as soon as you can
  - Limiting exposure
- Use tools like `chroot()` to compartmentalize

`chroot()`

- Unix command to set up sandboxed environment
- Programs run `chroot()` see different directory as the root of the file system
- Thus, can't see anything not under that directory
- Hard to set up right, though
- Other systems have different approaches