

CIS/TCOM 551

Computer and Network Security

Spring 2005

Lecture 2

So far...

- We've had some experience with:
 - Buffer overflows (in project 1)
 - Networks & network security (e.g. security in TCP/IP)
 - Basic cryptography (e.g. RSA and shared key cryptography)
- In project 2 you'll get some experience implementing software that uses cryptography (via OpenSSL)

How do you know it's secure?

- Clearly, appropriate use of cryptography is essential
 - Even that is hard to get right
- ...but even if cryptography is used appropriately, there are still plenty of possible vulnerabilities
 - e.g. buffer overflows (> 50% of CERT vulnerability)
 - 85% of CERT vulnerabilities could not be prevented with cryptography
- See the paper "Why Cryptosystems Fail" by Ross Anderson (available on course web page) for a nice discussion
 - A bit dated, because the paper was published in 1993
- Problem is bad software design and implementation.
- "Penetrate and Patch Model" doesn't work very well.

Building Secure Software

- Source: book by John Viega and Gary McGraw
 - Copy on reserve in the library
 - Strongly recommend buying it if you care about implementing secure software.
- Designing software with security in mind
- What are the security goals and requirements?
 - Risk Assessment
 - Tradeoffs
- Why is designing secure software a hard problem?
- Design principles
- Implementation
- Testing and auditing

Security Goals

- Prevent common vulnerabilities from occurring (e.g. buffer overflows)
- Recover from attacks
 - Traceability and auditing of security-relevant actions
- Monitoring
 - Detect attacks
- Privacy, confidentiality, anonymity
 - Protect secrets
- Authenticity
 - Needed for access control, authorization, etc.
- Integrity
 - Prevent unwanted modification or tampering
- Availability and reliability
 - Reduce risk of DoS

Other Software Project Goals

- Functionality
- Usability
- Efficiency
- Time-to-market
- Simplicity

- Often these conflict with security goals
 - Examples?

- So, an important part of software development is risk assessment/risk management to help determine the design choices made in light of these tradeoffs.

Risk Assessment

- Identify:
 - What needs to be protected?
 - From whom?
 - For how long?
 - How much is the protection worth?
- Refine specifications:
 - More detailed the better (e.g. "Use crypto where appropriate." vs. "Credit card numbers should be encrypted when sent over the network.")
 - How urgent are the risks?
- Follow good software engineering principles, but take into account malicious behavior.

Principles of Secure Software

- What guidelines are there for developing secure software?
- How would you go about building secure software?
Class answers:
 - Use a good programming language / compiler technology.
 - Protect against common forms of attacks.
 - Think about security from the start.
 - Minimize the trusted computing base.
 - Use try/catch to handle exceptions (think about failure).
 - Keep it simple.
 - Use well established technology (rather than starting from scratch)

#1: Secure the Weakest Link

- Attackers go after the easiest part of the system to attack.
 - So improving that part will improve security most.
- How do you identify it?
- Weakest link may not be a software problem.
 - Social engineering
 - Physical security
- When do you stop?

#2: Practice Defense in Depth

- Layers of security are harder to break than a single defense.
- Example: Use firewalls, and virus scanners, and encrypt traffic even if it's behind firewall

#3: Fail Securely

- Complex systems fail.
- Plan for it:
 - Aside: For a great example, see today's colloquium speaker George Candea who's Ph.D. research is about something called "microreboots"
- Sometimes better to crash or abort once a problem is found.
 - Letting a system continue to run after a problem could lead to worse problems.
 - But sometimes this is not an option.
- Good software design should handle failures gracefully
 - For example, handle exceptions

#4: Principle of Least Privilege

- Recall the Saltzer and Schroeder article
- Don't give a part of the system more privileges than it needs to do its job.
 - Classic example is giving root privileges to a program that doesn't need them: mail servers that don't relinquish root privileges once they're up and running on port 25.
 - Another example: Lazy Java programmer that makes all fields public to avoid writing accessor methods.
- Military's slogan: "Need to know"

#5: Compartmentalize

- As in software engineering, modularity is useful to isolate problems and mitigate failures of components.
- Good for security in general: Separation of Duties
 - Means that multiple components have to fail or collude in order for a problem to arise.
 - For example: In a bank the person who audits the accounts can't issue cashier's checks (otherwise they could cook the books).
- Good examples of compartmentalization for secure software are hard to find.
 - Negative examples?

#6: Keep it Simple

- KISS: Keep it Simple, Stupid!
- Einstein: "Make things as simple as possible, but no simpler."
- Complexity leads to bugs and bugs lead to vulnerabilities.
- Failsafe defaults: The default configuration should be secure.
- Ed Felten quote: "Given the choice between dancing pigs and security, users will pick dancing pigs every time."

#7: Promote Privacy

- Don't reveal more information than necessary
 - Related to least privileges
- Protect personal information
 - Consider implementing a web pages that accepts credit card information.
 - How should the cards be stored?
 - What tradeoffs are there w.r.t. usability?
 - What kind of authentication/access controls are there?

#8: Hiding Secrets is Hard

- The larger the secret, the harder it is to keep
 - That's why placing trust in a cryptographic key is desirable
- Security through obscurity doesn't work
 - Compiling secrets into the binary is a bad idea
 - Code obfuscation doesn't work very well
 - Reverse engineering is not that difficult
 - Software antipirating measures don't work
 - Even software on a "secure" server isn't safe (e.g. source code to Quake was stolen from id software)
 -

#9: Be reluctant to trust

- Trusted Computing Base: The set of components that must function correctly in order for the system to be secure.
- The smaller the TCB, the better.
- Trust is transitive
- Be skeptical of code quality
 - Especially when obtained from elsewhere
 - Even when you write it yourself

#10: Use Community Resources

- Software developers are not cryptographers
 - Don't implement your own crypto
 - (e.g. bugs in Netscape's storage of user data)
- Make use of CERT, Bugtraq, developer information, etc.

Discussion: Open Source

- Is Open Source Software more secure than proprietary software?