Principles and their Use

Matt Bishop
Dept. of Computer Science
University of California, Davis
One Shields Ave.
Davis, CA 95616-8562

email: bishop@cs.ucdavis.edu

Failure comes only when we forget our ideals and objectives and principles.

— Jawaharal Nehru

Outline

- Definition of "secure" software
- Quick review of Saltzer and Schroeder's principles
- Example of applying one to software development
- Some thoughts on developing curricula

What is “Secure” Software?

- CBK titled: “Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software”
- Security involves systems and system environments, not just software

In This Talk …

- “Secure software” means:
  1. The set of security requirements that the software is to satisfy is complete; and
  2. The software is developed, deployed, and operated in a manner that provides sufficient assurance to assert that the software satisfies the security requirements.

Assurance

Definition 18-1. An entity is *trustworthy* if there is sufficient credible evidence leading one to believe that the system will meet a set of given requirements. *Trust* is a measure of trustworthiness, relying on the evidence provided.

Definition 18-2. *Security assurance*, or simply *assurance*, is confidence that an entity meets its security requirements, based on specific evidence provided by the application of assurance techniques.

— *Computer Security: Art and Science* (p. 478)

What Is a Principle?

- A fundamental truth or proposition that serves as the foundation for a system of belief or behavior or for a chain of reasoning
  - A general scientific theorem or law that has numerous special applications across a wide field.
  - A natural law forming the basis for the construction or working of a machine : these machines all operate on the same general principle.

Why Saltzer and Schroeder?

- Widely recognized as good principles
- Applicable to wide variety of situations, systems, and abstractions
- Not bound to any particular technology

The Principles

- Least Privilege
- Fail-Safe Defaults
- Economy of Mechanism
- Complete Mediation
- Open Design
- Separation of Privilege
- Least Common Mechanism
- Psychological Acceptability

Least Privilege

- A subject should be given only those privileges necessary to complete its task
  - Function, not identity, controls
  - Rights added as needed, discarded after use
  - Minimal protection domain

Fail-Safe Defaults

- Default action is to deny access
- If action fails, system as secure as when action began

Economy of Mechanism

- Keep it as simple as possible
  - KISS Principle; easier to analyze ("analyzability")
- Simpler means less can go wrong
  - And when errors occur, they are easier to understand and fix
- Interfaces and interactions

Complete Mediation

- Check every access
- Usually done once, on first action
  - UNIX: access checked on open, not checked thereafter
- If permissions change after, may get unauthorized access

Open Design

- Security should not depend on secrecy of design or implementation
  - Popularly misunderstood to mean that source code should be public
  - "Security through obscurity"
  - Does not apply to information such as passwords or cryptographic keys

Separation of Privilege

- Require multiple conditions to grant privilege
  - Separation of duty
  - Defense in depth
  - Use standard functions; avoid writing your own!

Least Common Mechanism

- Mechanisms should not be shared
  - Information can flow along shared channels
  - Covert channels
- Isolation
  - Virtual machines
  - Sandboxes

Psychological Acceptability

- Security mechanisms should not add to difficulty of accessing resource
  - Hide complexity introduced by security mechanisms
  - Ease of installation, configuration, use
  - Human factors critical here

Those are my principles, and if you don't like them... well, I have others.

— Groucho Marx

Applying These

- Example here!
- Use "Principle of Psychological Acceptability" to develop points for secure software construction (Section 7 of SwACBK)
- What follows is not rigorous, but this technique can be made more rigorous than shown here

Basic Implications

- Make interfaces easy to use, hard to misuse
  - User interfaces
  - APIs, other internal interfaces
- Build using existing libraries and services
- Assume users, programmers will make mistakes

User Interfaces

- Create "mental model" of users
  - What does user expect, and how will she approach using the program/system?
  - What would confuse user?
    - Confusing or conflicting names …
  - Common error: thinking the user is a computer programmer or sophisticated
  - Example: electronic voting systems

Programmer Interfaces

- Look for confusing library, function interfaces
  - Inconsistencies in several similar interfaces (for example, arguments in different orders) or in use of arguments
- Assume other programmers will modify code
- Confusing or conflicting (operator) names

These Lead To …

- Validate *all* inputs
  - Assume malevolence and error
  - Validation depends on what is done with inputs
  - Do *not* assume correct configurations!

Language Selection

- Use an appropriate programming language
  - “Appropriate” means suitable for the problem
  - Consider security as one aspect of the problem
    - Some languages provide better security features than others
- Use appropriate libraries, too!

Document

- Comments must match code
- Comments should be intelligible for the audience
  - For programmers, document interfaces and choice of algorithms
  - For administrators, document configuration parameters

Elements of Best Practices (§7.4)

- Minimize code size and complexity, and increase traceability: this will make the code easy to analyze
- Code with reuse and sustainability in mind: this will make code easy to understand by others
- Use a consistent coding style throughout the system: this is the objective of the coding standards described in subsection 7.2.4.
- Make security a criterion when selecting programming languages to be used
- ... use input validation, compiler checks to verify correct language usage and flag “dangerous” constructs, ... absence of “dangerous” constructs and characters
- Use consistent naming and correct encapsulation
- Implement error and exception handling safely
- Program defensively: Use techniques such as information hiding and anomaly awareness
- Always assume that the seemingly impossible is possible: the history of increased sophistication, technical capability, and motivation of attackers shows that events, attacks, and faults that seem extremely unlikely when the software is written can become quite likely after it has been in operation for a while. Error and exception handling should be programmed explicitly to deal with as many “impossible” events as the programmer can imagine
- ... this includes never trusting parameters passed by the environment ... always presuming client/user hostility (thus always validating all input from the client/user)

General Rules

- Paranoia
  - They are out to get you!
- Stupidity
  - If something can be misinterpreted, it will be
- Can’t happen
  - Someone makes a change … now it can

Using This For Curricula

- Develop commonly agreed-upon principles
- From those, develop several types of curricula
  - Training
  - Undergraduate education
  - Graduate education

Training

- Focus on particular systems, situations, or both
- Typically intensive, hands-on
- What you learn immediately applicable to particular systems, situations
  - May not generalize beyond that

For This Domain

- Develop guidelines, “best practices” for particular environments from the principles
- Map principles into these and verify:
  - Guidelines come from principles
  - Principles that are not reflected in the guidelines don’t apply
- *Very* specific to environment, systems, *etc.*

Undergraduate Education

- Learn broad principles and their application
  - Does *not* focus on particular system, environment
- Use case studies to illustrate, guide students to understanding principles and how to apply them in other, new environments
  - And learn what questions to ask …

Master's Education

- Expands on principles taught in undergraduate education
- Learn how to analyze problems and proposed solutions in depth
- Learn how to identify, balance competing interests and how to apply different technologies to achieve balance

Doctoral Education

- Also builds on undergraduate education
- Analyze principles and extend, change, or improve them
  - Or derive new ones
- Emphasizes fundamental research pushing boundaries of knowledge
  - Results may not be immediately applicable

Applying Principles

- Do not tie teaching principles into any particular problem domain
- Curricula must be developed from principles to provide a coherent, organized foundation to the subject
- One curriculum will fail to meet the disparate needs of students, employee roles, *and employers*

Questions About the SwACBK

- How do you decide what "security" means?
  - Are your mom's security needs the same as mine?
- In general, separate discussions of *functionality* from *assurance*
- Organize chapters around principles
  - Leads to good classifications
- Expand discussion of principles, concepts
  - Much more on reference monitors, etc.
- Choose original reference sources
  - Say why each reference is there

Conclusion

- Curricula in academia are best when based on principles
  - Not tied to a specific domain area
  - Instantiated to domain area as needed
- SwACBK is not structured to drive curricula development in academia, nor can it be in its current form

How To Do This

- Workshop to develop principles
  - Need broad community acceptance to have an effect
- Work with academics who teach in the different educational environments to develop several model curricula for different types of education and institutions

Accept that increasing strength in software development will require a long-term effort and not have an immediate pay-off and require focusing on principles, not mechanics!

The mechanics will change with changes in environment and technology

We need people who will be able to adapt to these changes, and develop guidelines that reflect the changes; this includes re-education

Changing a college curriculum is like moving a graveyard—you never know how many friends the dead have until you try to move them!

— Woodrow Wilson