Security Principles

Secure Programming
Lecture 2

Announcement

- Homework 1 is out!
  http://www.cs.bham.ac.uk/~covam/teaching/2013/secprog/hw1.html
- Due date:
  Tuesday, January 21, at 11:59pm UTC

Running example

A. Barth et al., The Security Architecture of the Chromium Browser, 2008

Key terms

Bug:

- flaw in a program that results in unexpected behavior

Vulnerability:

- a bug with security-relevant consequences

Exploit:

- code that leverages a vulnerability to compromise the security of a system

System’s security

- Expressed in terms of a *security policy*
- List of actions that are permitted and behaviors that should be forbidden
- Most often informal; in certain domains (e.g., credit card processing) explicitely expressed
- What about formal policies?

Security expectations

Security policies are most often concerned with:

- Confidentiality
- Integrity
- Availability

Risk

risk = f(threat, vulnerability, likelihood, impact)

(Entire course could be done on risk!
If you want to know more, NIST's Guide for Conducting Risk Assessments is a good starting point)

Types of vulnerabilities

Design: flaw in the design
Implementation: error in how the system is written
Operational: issue in how the system is used

Taxonomies of vulnerabilities (read more)

Why do we classify vulnerabilities?

- cost of fixing vulnerabilities
- predicting vulnerabilities

Why so many vulnerabilities?

Complexity

- Code/design is too complex to understand all its implications, relationships, and assumptions
- Maybe it's not sufficiently documented

Dijkstra, Programming is hard

Why so many vulnerabilities?

(Lack of) Education

- Developers may not know about security issues
- That's one of the reasons why you're here!

Why so many vulnerabilities?

Extensibility

- What your system is supposed to do changes over time
- The assumptions about the way your system is going to be used change over time

Why so many vulnerabilities?

(Lack of) time

- Your product launches in 2 weeks: you can fix the vulnerabilities and be late or ignore them and ship on time...

Secure design principles

Saltzer and Schroeder, The Protection of Information in Computer Systems, 1975

- Difficult to design and implement secure systems
- Systematic, methodical techniques are not available to do so
- But experience provides useful set of principles

Adi Shamir, “There are no secure systems, only degrees of insecurity”

For a review of the original paper, see R. Smith,
A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles

Economy of mechanism

Keep the design as simple and small as possible.

- simple != small
- interactions are hard (need to check how each subset interact with others)

Security audits necessary and they are only successful on small and simple systems

Complete mediation

Every access to every object is checked for permission.

- “every access”: caching of permission check results?
- “check for permission”: authentication + authorization

Fail-safe defaults

Base security decisions on permission rather than exclusion.

Deny as default (good)

- Grant access only on explicit permission
- Mistakes leads to false negatives (access denied to legitimate user): quickly reported
- Denial of service?

Sometimes called “whitelisting” (input validation)

Fail-safe defaults

Base security decisions on permission rather than exclusion.

Allow as default (bad)

- Grant access when not explicitly prohibited
- Mistakes leads to false positives (access allowed to bad users): they don't tend to report…
- Hard to consider all corner cases/future cases
- Wrong mindset
- Ease of use

Sometimes called “blacklisting” (input validation)

Open design

Security of the system must not depend on the secrecy of its design (known since 1853).

Advantages of openness:

- enables the review of mechanisms by other experts
- establishes trust
- forces correct mindset/psychology on developers

Possible to keep secrecy in widely distributed systems?

What about the price of attacks? Risk of being detected?

Does being open automatically make you secure?

Separation of privileges

Make access depend on more than one condition.

- For example, two keys are needed to access a resource
- Privileges can be separated
- More than one attack is needed to compromise the system

Practical examples:

- Something you know, something you have, something you are
- 2-factor authentication in banks (?) and Google

Separation of privileges

Related concept: *compartmentalization*

- divide system in different, isolated parts
- minimize privileges of each
- don't implement all-or-nothing model
- → minimizes possible damage

Sandbox:

- Virtual machines
- Java sandbox (bytecode verifier, class loader, security manager)
- Janus (research project)

Least privilege

Operate using only the least set of privileges necessary.

- Minimize damage
- Minimize interaction between privileged programs

Interesting cases:

- setuid root programs (UNIX)
- database access

Least privilege

Corollaries:

- Minimize time that privilege can be used (drop privileges as soon as possible)
- Minimize time privilege is active (temporarily drop privileges)
- Minimize components that are granted privilege
- Minimize resources that are available to privileged program (e.g., chroot, jail mechanisms, quota)

Example: OpenSSH

- N. Provos et al., Preventing Privilege Escalation, USENIX Security, 2003

Least privilege

Implementation:

- split application into smaller protection domains (compartments)
- assign right privileges to each compartment
- devise channels between compartments
- ensure that channels remain isolated, except for how intended
- make it easy to audit

Sounds complicated, isn't it?

How do you know the set of privileges/capabilities that are required? Technique: start with none and add

Least common mechanism

Minimize the amount of mechanisms shared between and relied on by multiple users.

- Reduce potentially dangerous information flow
- Reduce unintended interactions
- Minimize consequences of vulnerabilities found in a mechanism

Software homogeneity and its consequences

Psychological acceptability

User interface must be easy to use.

- Ease of applying mechanism routinely and correctly
- Password change policies and sticky notes
- Firewall policies and bring-your-own-modems

User interface must conform to user's mental model:

- reduce likelihood of mistakes

Circumvention work factor

Security = f(cost of circumvention).

- Resources available to adversary?
- Cost of using those resources?
- It makes sense to focus on increasing the cost of exploiting bug, rather than on discovering new bugs

Example: password breaking or secret key brute-forcing

"Security is economics"

Compromise recording

Sometimes it sufficient to know that a system has been compromised.

- Tamperproof logging
- Intrusion Detection Systems (IDSes)

"If you can't prevent, detect"

Other principle: orthogonal security

Sometimes security mechanisms can be implemented orthogonally to the systems they protect.

- Simpler
- Applicable to legacy code
- Can be composed into multiple layers (“Defense in depth”)

Examples: security wrappers, IDSes, etc.

Other principle: be skeptical and be paranoid

Skeptical: force people to justify security declarations

Paranoid: “Never underestimate the amount of time and effort that someone will put into breaking your system” (Robert Morris)

Other principle: design security in

Security must be a key design factor.

- Applying these principles is not easy when you start off with the intention to do so, imagine if you have to retrofit a system that was not designed with them in mind

Take away points

- Designing and building secure systems is hard (for many reasons)
- Set of principles help us doing that (and evaluating existing systems)

Next time

- Finding vulnerabilities