CS412 Software Security

Basic Principles

Mathias Payer

EPFL, Spring 2019

Software Security

Allow intended use of software, prevent unintended use that may cause harm.

Security principles

- **Confidentiality:** an attacker cannot recover protected data
- **Integrity:** an attacker cannot modify protected data
- **Availability:** an attacker cannot stop/hinder computation

Accountability/non-repudiation may be used as fourth fundamental concept. It prevents denial of message transmission or receipt.

Security analysis

Given a software system, is it secure?

- . . . it depends
- What is the attack surface?
- What are the assets? (How profitable is an attack?)
- What are the goals? (What drives an attacker?)

Attacks and Defenses

- Attack (threat) models
    - A class of attacks that you want to stop
        - What is the attacker's capability?
        - What is impact of an attack?
        - What attacks are out-of-scope?
- Defenses address a certain attack/threat model.
    - General: e.g., stop memory corruptions
    - Very specific: e.g., stop overwriting a return address

Threat model

The threat model defines the abilities and resources of the attacker. Threat models enable structured reasoning about the attack surface.

- Awareness of entry points (and associated threats)
- Look at systems from an attacker's perspective
  - Decompose application: identify structure
  - Determine and rank threats
  - Determine counter measures and mitigation
- Reading material: `https://www.owasp.org/index.php/Application_Threat_Modeling`

Threat model: safe

Assume you want to protect your valuables by locking them in a safe.

- In trust land, you don't need to lock your safe.
- An attacker may pick your lock.
- An attacker may use a torch to open your safe.
- An attacker may use advanced technology (x-ray) to open it.
- An attacker may get access (or copy) your key.

Threat model: operating systems

- *Malicious extension:* inject an attacker-controlled driver into the OS;
- *Bootkit:* compromise the boot process (BIOS, boot sectors);
- *Memory corruption:* software bugs such as spatial and temporal memory safety errors or hardware bugs such as rowhammer;
- *Data leakage:* the OS accidentally returns confidential data (e.g., randomization secrets);
- *Concurrency bugs:* unsynchronized reads across privilege levels result in TOCTTOU (time of check to time of use) bugs;
- *Side channels:* indirect data leaks through shared resources such as hardware (e.g., caches), speculation (Spectre or Meltdown), or software (page deduplication);
- *Resource depletion and deadlocks:* stop legitimate computation by exhausting or blocking access to resources

Cost of security

There is no free lunch, security incurs overhead.
Security is. . .

- expensive to develop,
- may have performance overhead,
- may be inconvenient to users.

Fundamental security mechanisms

- Isolation
- Least privilege
- Fault compartments
- Trust and correctness

Figure 1

Isolation

Isolate two components from each other. One component cannot access data/code of the other component except through a well-defined API.

Least privilege

The principle of least privilege ensures that a component has the least privileges needed to function.

- Any privilege that is further removed from the component removes some functionality.
- Any additional privilege is not needed to run the component according to the specification.
- Note that this property constrains an attacker in the privileges that can be obtained.

Fault compartments

> *Separate individual components into smallest functional entity possible. General idea: contain faults to individual components. Allows abstraction and permission checks at boundaries.*

Note that this property builds on least privilege and isolation. Both properties are most effective in combination: many small components that are running and interacting with least privileges.

Fault compartments

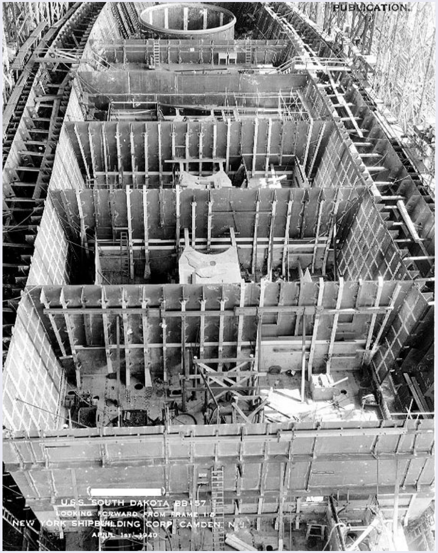
PUBLICATION.
U.S.S. SOUTH DAKOTA BB-57
LOOKING FORWARD FROM FRAME 118
NEW YORK SHIPBUILDING CORP. CAMDEN, N.J.
APRIL 1st 1940

Figure 2

Trust and correctness

Specific components are assumed to be trusted or correct according to a specification.

Formal verification ensures that a component correctly implements a given specification and can therefore be trusted. Note that this property is an ideal property that cannot generally be achieved.

Hardware and software abstractions

- Operating System (OS) abstractions
- Hardware abstractions

Operating System (OS) abstraction

- Provides process abstraction
- Well-defined API to access hardware resources
- Enforces mutual exclusion to resources
- Enforces access permissions for resources
- Restrictions based on user/group/ACL
- Restricts attacker

OS process isolation

- Memory protection: protect the memory (code and data such as heap, stack, or globals) of one process from other processes
- Address space: working memory of one process
- Today's system implement address spaces (virtual memory) through page tables with the help of an Memory Management Unit (MMU)

OS designs: single domain (1/4)

- A single layer, no isolation or compartmentalization
- All code runs in the same domain: the application can directly call into operating system drivers
- High performance, often used in embedded systems

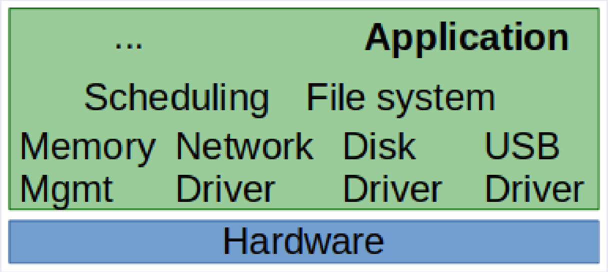

Figure 3

OS design: monolithic (2/4)

- Two layers: the operating system and applications
- The OS manages resources and orchestrates access
- Applications are unprivileged, must request access from the OS
- Linux fully and Windows mostly follows this approach for performance (isolating individual components is expensive)

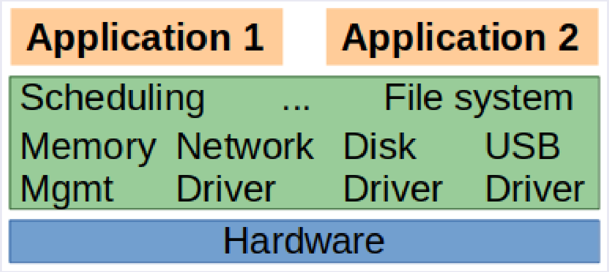

Figure 4

OS design: micro-kernel (3/4)

- Many layers: each component is a separate process
- Only essential parts are privileged
  - Process abstraction (address spaces)
  - Process management (scheduling)
  - Process communication (IPC)
- Applications request access from different OS processes

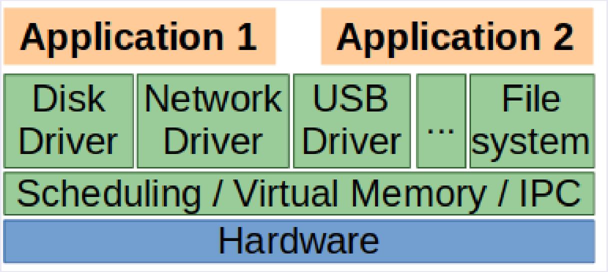

Figure 5

OS design: library os (4/4)

- Few thin layers; flat structure
- Micro-kernel exposes bare OS services
- Each application brings all necessary OS components

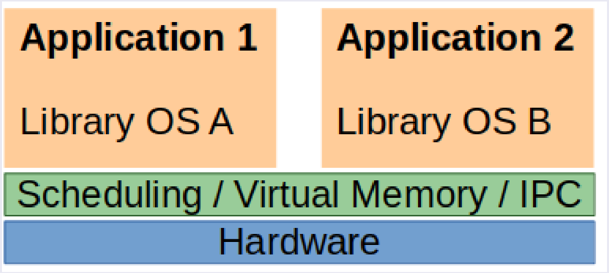

Figure 6

Hardware abstraction

- Virtual memory through MMU/OS
- Only OS has access to raw physical memory
- DMA for trusted devices
- ISA enforces privilege abstraction (ring 0/3 on x86)
- *Hardware abstractions are fundamental for performance*

Access control

- **Authentication:** Who are you (what you know, have, or are)?
- **Authorization:** Who has access to object?
- **Audit/Provenance:** I'll check what you did.

Authentication: who are you?

There are three fundamental types of identification:

- What you know: username / password
- What you are: biometrics
- What you have: second factor / smartcard

How do you authenticate a remote entity?

- Kerberos: capabilities and tokens.

Authorization: Information flow control

An important question when handling shared resources is who can access what information.

- These access policies are called access control models.
- Access control models were originally developed by the US military
  - Users with different clearance levels on a single system
  - Data was shared across different levels of clearance
  - Therefore the name “multi-level security”

Authorization: Types of access control

- Mandatory Access Control (MAC): Rule and lattice-based policy
- Discretionary Access Control (DAC): Object owners specify policy
- Role-Based Access Control (RBAC): Policy defined in terms of roles (sets of permissions), individuals are assigned roles, roles are authorized for tasks.

MAC

- Centrally controlled
- One entity controls what permissions are given
- Users cannot change policy themselves
- Examples: Bell/LaPadula and Biba

Bell and LaPadula

- Multi level security model that enforces information flow control
- All security levels are monotonically ordered, files have clearance level
- A given clearance allows reading files of lower or equal clearance and writing files of equal or higher clearance.
- Summary: read-down, write-up
- Bell/LaPadula enforces *confidentiality*

Biba

- Multi level security model that enforces information flow control
- All security levels are monotonically ordered, files have integrity level
- A given clearance allows reading files of higher or equal clearance and writing files of lower or equal clearance.
- Summary: read-up, write-down
- Biba enforces *integrity*

DAC

- MAC is complex and requires central control, empower the user!
- User has authority over resources she owns
- User determines permissions for her data if other users want to access it
- For example: Unix permissions

RBAC

- Access permission is broken into sets of roles.
- Users get assigned specific roles
- Administration privileges may be a role.

Different security models

- Access control lists (static)
- Capabilities (static)
- Bell-LaPadula (state machine)
- Information flow (flow dependent)

Access control matrix

Provide access rights for subjects to objects.

	foo	bar	baz
user	rwx	rw	rw
group	rx	r	r
other	rx		r

- Used, e.g., for Unix/Linux file systems or Android/iOS/Java security model for privileged APIs.
- Introduced by Butler Lampson in 1971 `http://research.microsoft.com/en-us/um/people/blampson/08-Protection/Acrobat.pdf`.

Summary and conclusion

- Software security goal: allow intended use of software, prevent unintended use that may cause harm.
- Three principles: Confidentiality, Integrity, Availability.
- Security of a system depends on its threat model.
- Isolation, least privilege, fault compartments, and trust as concepts.
- Security relies on abstractions to reduce complexity.
- Reading assignment: Butler Lampson, Protection `http://doi.acm.org/10.1145/775265.775268`