2020 T2 Week 07a
Security Fundamentals
@GernotHeiser
Incorporating material from Toby Murray
Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    “Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
What is Security?

Different things to different people:

On June 8, as the investigation into the initial intrusion proceeded, the response team shared with relevant agencies that there was a high degree of confidence that OPM systems containing information related to the background investigations of current, former, and prospective Federal government employees, and those for whom a Federal background investigation was conducted, may have been compromised.
Computer Security

Protecting *my interests* (that are under computer control) from *threats*

- Inherently subjective
  - Different people have different interests
  - Different people face different threats
- Don’t expect one-size-fits-all solutions
  - Grandma doesn’t need an air gap
  - Windows insufficient for protecting TOP SECRET (TS) classified data on an Internet-connected machine

Security claims only make sense
- *wrt* defined objectives
- *while* identifying threats
- *and* identifying secure states
State of OS Security

• Traditionally:
  • Has not kept pace with evolving user demographics
    • Focused on e.g. Defence and Enterprise
  • Has not kept pace with evolving threats
    • Much security work is reactive rather than proactive

Some things are getting better:
• more systematic hardening of OSes
• Better security models in smartphones compared to desktops

Other things are getting worse:
• OS kernel sizes keep growing
• Fast growth in attacker capabilities
• Slow growth in defensive capabilities
OS Security

• What is the role of the OS for security?
• Minimum:
  • provide **mechanisms** to allow the construction of secure systems
  • that are capable of securely implementing the intended users’/administrators’ **policies**
  • while ensuring these mechanisms cannot be subverted
Good Security Mechanisms

• Are widely applicable
• Support general security principles
• Are easy to use correctly and securely
• Do not hinder non-security priorities (e.g. productivity, generativity)
  • Principle of “do not pay for what you don’t need”

Good mechanisms lend themselves to correct implementation and verification!
Security Design Principles

Saltzer & Schroeder [SOSP ’73, CACM ’74]

- **Economy of mechanism** – KISS
- **Fail-safe defaults** – as in any good engineering
- **Complete mediation** – check everything
- **Open design** – not security by obscurity
- **Separation of privilege** – defence in depth
- **Least privilege** – aka *principle of least authority* (POLA)
- **Least common mechanism** – minimise sharing
- **Psychological acceptability** – if it’s hard to use it won’t be
Common OS Security Mechanisms

- Access Control Systems
  - control what each process can access
- Authentication Systems
  - confirm the identity on whose behalf a process is running
- Logging
  - for audit, detection, forensics and recovery
- Filesystem Encryption
- Credential Management
- Automatic Updates
Security Policies

- Define what should be protected, and from whom

- Often in terms of common security goals (*CIA properties*):
  - **Confidentiality**
    - $X$ should not be learnt by Low
  - **Integrity**
    - $Y$ should not be tampered with by Low
  - **Availability**
    - $Z$ should not be made unavailable to High by Low
Security vs Safety

Fundamentally, OS-level security & safety enforcement is about isolation.
Policy vs Mechanism

• Policies accompany mechanisms:
  • access control policy
    • who can access what?
  • authentication policy
    • is password sufficient to authenticate TS access?
• Policy often restricts the applicable mechanisms
• One person’s policy is another’s mechanism
Assumptions

• All policies and mechanisms operate under certain **assumptions**
  • *e.g.* TS-cleared users can be trusted not to write TS data into the UNCLASS window

• Problem: implicit or poorly understood assumption

---

**Good assumptions are**

• *clearly identified*
• *verifiable!*
Risk Management

• Comes down to **risk management**
  • There is no absolute security, what risks we are willing to tolerate?
  • Cost & likelihood of violation vs. cost of prevention
  • Gain vs cost for attacker

• Actions:
  • mitigate – using security mechanisms
  • transfer – e.g. by buying insurance

Good security policy will identify appropriate action, based on risk assessment
Trust

- Systems always have **trusted** entities
  - whose misbehaviour can cause insecurity
  - hardware, OS, sysadmin ...

- Secure systems require the TCB to be **trustworthy**
  - achieved through **assurance** and **verification**
  - shows that the TCB is unlikely to misbehave

*Minimising the TCB is key* for ensuring correct behaviour

**Trusted computing base (TCB):**
The set of all trusted entities
Assurance and Formal Verification

- **Assurance:**
  - systematic evaluation and testing
  - essentially an intensive and onerous form of quality assurance

- **Formal verification:**
  - mathematical proof

- **Certification:** independent examination
  - confirming that the assurance or verification was done right
Covert Channels

• Information flow not controlled by security mechanisms
  • Confidentiality requires absence of all such channels
• **Storage** Channel: Attribute of shared resource used as channel
  • Controllable by access control
• **Timing** Channel: Temporal order of shared resource accesses
  • Outside of access-control system
  • Much more difficult to control and analyse
• Other **physical** channels:
  • Power draw
  • Temperature (fan speed)
  • Electromagnetic emanation
  • Acoustic emanation

```c
void leak(secret){
    if (secret) {
        create (“/tmp/true”);
    } else {
        create (“/tmp/false”);
    }
}
```
Covert Timing Channels

- Created by shared resource whose effect on timing can be monitored
  - network bandwidth, CPU load, memory latency ...
- Requires access to a time source
  - Anything that allows processes to synchronise
  - Generally any relative occurrence of two event
- Critical issue is channel bandwidth
  - low bandwidth limits damage
    - why DRM ignores low bandwidth channels
  - beware of amplification
    - e.g. leaking passwords, encryption keys etc.

Typical timing channels:
- Measure server response times
- Measure own progress
Covert Channels vs Side Channels

**Covert Channel**
- Trojan intentionally creates signal through targeted resource use
- Worst-case bandwidth

**Side Channel**
- Attacker uses signal created by victim’s innocent operations
- Much lower bandwidth
Summary of Introduction

• Security is very subjective, needs well-defined objectives
• OS security:
  • provide good security mechanisms
  • that support users’ policies
• Security depends on establishing trustworthiness of trusted entities
  • TCB: set of all such entities
    • should be as small as possible
  • Main approaches: assurance and verification

The OS is necessarily part of the TCB
Access-Control Principles
Access Control

**Who** can access **what** in which ways

- The “who” are called **subjects** (or **agents**)
  - e.g. users, processes etc.
- The “what” are called **objects**
  - e.g. individual files, sockets, processes etc.
  - includes all subjects
- The “ways” are called **permissions**
  - e.g. read, write, execute etc.
  - are usually specific to each kind of object
  - include those meta-permissions that allow modification of the protection state
    - e.g. own
Access Control Mechanisms & Policies

• Access Control Policy
  • Specifies allowed accesses
  • And how these can change over time

• Access Control Mechanism
  • Used to implement the policy

• Certain mechanisms lend themselves to certain kinds of policies
• Some policies cannot be expressed using your OS’s mechanisms
Protection State: Access-Control Matrix

Defines system’s protection state at a particular time instance [Lampson ‘71]

<table>
<thead>
<tr>
<th>Subj</th>
<th>Obj1</th>
<th>Obj2</th>
<th>Obj3</th>
<th>Subj2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subj1</td>
<td>R</td>
<td>RW</td>
<td></td>
<td>send</td>
</tr>
<tr>
<td>Subj2</td>
<td></td>
<td>RX</td>
<td></td>
<td>control</td>
</tr>
<tr>
<td>Subj3</td>
<td>RW</td>
<td></td>
<td>RWX</td>
<td>own</td>
</tr>
</tbody>
</table>

Subjects are also objects
Storing full matrix too inefficient
- huge but sparse
- highly dynamic

Store by row or by column

Obj1
Subj1: R
Subj3: RW

Obj2
R
RW
RX

Obj3
RW
RWX
own
recv

Subj2
send
control
recv

Subj3
Obj1: RW
Obj3: RWX, own
Subj2: recv

Access-control list (ACL)

Capability list (Clist)

Defines subject’s protection domain

© Gernot Heiser 2019 – CC Attribution License
Access Control Lists (ACLs)

- Subjects usually aggregated into classes
  - e.g. UNIX: owner, group, everyone
  - more general lists in Windows, recent Linux
- Can have negative rights
  eg. to overwrite group rights
- Meta-permissions (e.g. own)
  - control class membership
  - allow modifying the ACL

Used by all mainstream OSes
Capability-Based Access Control

Capability = Access Token:
Prima-facie evidence of privilege

Object reference
Access rights

Capabilities provide:
- Fine-grained access control
- Reasoning about information flow

Any system call is invoking a capability:
\( \text{err} = \text{cap.method}(\text{args}) \);

Used in very few commercial systems:
- IBM System/38 → AS/400 → i-Series
- KeyKOS [Bomberger et al, 1992]
Capabilities: Implementations

• Capabilities must be unforgeable
  • Traditionally protected by hardware (tagged memory), eg System-38
  • Can be copied etc like data
  • eg IBM System/38, Hydra, Cheri

• On conventional hardware, either:
  • Stored as ordinary user-level data, but unguessable due to sparseness
    • contains password or secure hash: PCS [Anderson’86], Mungi
    • “sparse” capabilities
  • Privileged kernel data
    • referred to by user programs by index/address
    • eg Mach [Accetta’86], EROS [Shapiro’99], seL4, Unix file descriptors
    • “partitioned” or “segregated” capabilities
ACLs & Capabilities – Duals?

- In theory dual representations of access control matrix
- Practical differences:
  - Naming and namespaces
    - Ambient authority
    - Deputies
  - Evolution of protection state
  - Forking
  - Auditing of protection state
Duals: Naming and Name Spaces

• ACLs:
  • objects referenced by **name**
    • requires separate (global) name space
    • e.g. `open("/etc/passwd",O_RDONLY)`
  • require a subject (class) namespace
    • e.g. UNIX users and groups

• Capabilities:
  • objects referenced by **capability**
  • no further namespace required
  • cannot even **name** object without access

Cover storage channel?
Duals: Confused Deputy

- **Subject**: Alice
- **Deputy**: `gcc`
- **Log file**: `/var/gcc/log`

Alice execs `gcc -o LogFile source.c`

**Code snippet**:
```c
static char* log = "/var/gcc/log";
int gcc (char *src, *dest) {
    int s = open (src, RDONLY );
    int l  = open (log, APPEND);
    int d = open (dest, WRONLY);
    ...  
    write (dest, ...);
}
```

- **Unix**: Log file is group `admin`
- **Alice**: not member of `admin`
- **gcc**: set-UID as `admin`

- **ACLs**: separate naming and permissions
- **Deputy**: depends on *ambient authority*
  - Uses own authority for access

*Confused-deputy problem is unsolvable with ACLs!*

Clobber log!
Duals: Confused Deputy

Subject: Alice

Deputy: gcc

Log file

Alice$ gcc -o LogFile source.c

* Caps are both names and permissions
* Presentation is *explicit*, not ambient
  * Can’t name something if don’t have access!

Cap system:
* gcc holds \( w \) cap for log file
* Alice holds \( r \) cap for source, \( w \) cap for destination
* Alice holds no cap for log file

Open fails!

Capabilities avoid confused deputies

```c
static cap_t log = <cap>;
int gcc (cap_t src, dest) {
    fd_t s = open (src, RDONLY);
    fd_t l = open (log, APPEND);
    df_t d = open (dest, WRONLY);
    ...
    write (d, ...);
}
```
Duals: Evolution of Protection State

**ACLs**: Protection state changes by modifying ACLs
- Requires certain meta-permissions on the ACL

**Capabilities**: Protection state changes by delegating and revoking caps
- Fundamental properties enable reasoning about *information flow*:
  - A can send message to B only if A holds cap to B
  - A can obtain access to C only if it receives message with cap to C
- *Right to delegate* may also be controlled by capabilities, e.g.:
  - A can delegate to B only if A has a *delegatable* capability to B
  - A can delegate X to B only if it has *grant* authority on X
Duals: Process Creation

• What permissions should children get?
• ACLs: depends on the child’s subject
  • UNIX etc.: child inherits parent’s subject
    • Inherits all of the parent’s permissions
    • Any program you run inherits all of your authority
  • Opposite of least privilege!
• Capabilities: child has no caps by default
  • Parent gets a capability to the child upon fork
  • Used to delegate explicitly the necessary authority
  • Defaults to least privilege
Duals: Auditing of Protection State

• Who has permission to access a particular object (right now)?
  • ACLs: Just look at the ACL
  • Caps: hard to determine with sparse or tagged caps, or for partitioned

• What objects can a particular subject access (right now)?
  • Capabilities: Just look at its capabilities
  • ACLs: may be impossible to determine without full scan

“Who can access my stuff?” vs
“How much damage can C do?”
Interposing Access

Caps are opaque object references (pure names)
- Holder cannot tell which object a cap references nor the authority
- Supports transparent interposition (virtualisation)

Usage:
- API virtualisation
- Security monitor
  - Security policy enforcement
  - Info flow tracing
  - Packet filtering…
- Secure logging
- Debugging
- Lazy object creation
Example: Lazy Object Construction

Client

Server

obj() {
  = create...
  substitute cap
}

meth() {
  perf operation
}

obj1
obj2
obj3

obj1.meth(args);
...
obj1.meth(args);
### Duals: Satzer & Schroeder Principles

<table>
<thead>
<tr>
<th>Security Principle</th>
<th>ACLs</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Economy of Mechanism</td>
<td>Dubious</td>
<td>Yes!</td>
</tr>
<tr>
<td>Fail-safe defaults</td>
<td>Generally not</td>
<td>Yes!</td>
</tr>
<tr>
<td>Complete mediation</td>
<td>Yes (if properly done)</td>
<td>Yes (if properly done)</td>
</tr>
<tr>
<td>Open design</td>
<td>Neutral</td>
<td>Neutral</td>
</tr>
<tr>
<td>Separation of privilege</td>
<td>No</td>
<td>Doable</td>
</tr>
<tr>
<td>Least privilege</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Least common mechanism</td>
<td>No</td>
<td>Yes, but…</td>
</tr>
<tr>
<td>Psychological acceptability</td>
<td>Neutral</td>
<td>Neutral</td>
</tr>
</tbody>
</table>
Mandatory vs Discretionary Access Control

Discretionary Access Control (DAC):
- Users can make access control decisions
  - Delegate their access to other users etc.

Mandatory Access Control (MAC):
- System enforces administrator-defined policy
- Users can only make access control decisions subject to mandatory policy
- Can prevent untrusted applications from causing damage
- Traditionally used in national security environments

Can I stop my browser leaking secrets?
Bell & LaPadula (BLP) Model [1966]

- MAC Policy/Mechanism
  - Formalises national security classifications
- Every object assigned a **classification**
  - e.g. TS, S, C, U
  - orthogonal security **compartments**
    - Support need-to-know
- Classifications ordered in a **lattice**
  - e.g. TS > S > C > U
- Every subject assigned a **clearance**
  - Highest classification they’re allowed to learn
BLP: Rules

• **Simple Security Property** ("no read up"):  
  • s can read o iff clearance(s) >= class(o)  
  • s-cleared subject can read U,C,S but not TS  
  • standard confidentiality

• **★-Property** ("no write down"):  
  • s can write o iff clearance(s) <= class(o)  
  • S-cleared subject can write TS,S, but not C,U  
  • to prevent accidental or malicious leakage of data to lower levels

• In practice need exceptions.  
  • allow trusted entity to write down  
  • de-classify

E.g. logging
MAC With Caps

```
interpose_transfer(cap) {
    if (A.clear > B.clear) {
        c = mint(cap, -r);
        send(B,c);
    } else if (A.clear < B.clear) {
        c = mint(cap, -w);
        send(B,c);
    } else {
        send(B,cap);
    }
}
```
Biba Integrity Model

- Bell-LaPadula enforces **confidentiality**
- **Biba**: Its dual, enforces **integrity**
- Objects now carry **integrity** classification
- Subjects labelled by **lowest** level of data each subject is allowed to learn
- BLP order is inverted:
  - s can read o iff clearance(s) <= class(o)
  - s can write o iff clearance(s) >= class(o)
Confidentiality + Integrity

• BLP+Biba allows no information flow across classes
• Practicality requires weakening
  • Assume high-classified subject to treat low-integrity info responsibly
  • Allow read-down
• **Strong *-Property** (“matching writes only”):
  • s can write o iff clearance(s) = class(o)
  • Eg for logging, high reads low data and logs
Clark & Wilson Model

• In commercial settings integrity is more important than confidentiality
• Restrict possible operations to *well-formed transactions*
  • eg payment issued only after goods and invoice received
  • performed by trusted programs
  • easy with caps, SetUID cesspit with ACLs
• Restrict access to trusted programs to specific people
  • separation of duty: running payment program separate from goods-received role
Boebert’s Attack on Capability Machines

“On the inability of an unmodified capability machine to enforce the ★-property” [Boebert’84]

Takeaway: Need mechanism to limit cap propagation: take-grant model

Works where caps are indistinguishable from data (HW & sparse caps)
Decidability

**Safety**: Given initial *safe state* \( s \), system will never reach *unsafe state* \( s' \).

**Decidability**: AC system is decidable if safety can always be computationally determined.

Equivalent to halting problem [Harrison, Ruzzo, Ullman ‘75]

- Most capability systems are decidable
- Unclear for many common ACL systems
Summary: AC Principles

• ACLs and Capabilities:
  • Capabilities tend to better support least privilege
  • But ACLs can be better for auditing
• MAC good for global security requirements
• Not all mechanisms can enforce all policies
  • e.g. ★-property with sparse or HW capabilities
• AC systems should be decidable so we can reason about security