COMP9242 Advanced OS

S2/2016 W10: **Operating System Security**

@GernotHeiser

Incorporating Material from Toby Murray

Never Stand Still | Engineering | Computer Science and Engineering

Copyright Notice

What is security?

Different things to different people:

Computer Security

- Protecting **my** interests that are under computer control from malign 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 alone is insufficient for protecting TOP SECRET (TS) classified data on an Internet-connected machine

Claiming system "*security*" only makes sense with respect to well-defined *security objectives*:

- Identify *threats*
- Identify set of *secure system 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
    - Focused on protecting users from users, not apps they run
- Is getting better
  - Eg smartphone OSes implement stricter security than desktops
  - But is hindered because:
    - OSes are still getting larger and more complex
    - Too few people understand how to write secure code

  UNSW AUSTRALIA

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

  UNSW AUSTRALIA

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"
- Lend themselves to correct implementation and verification

  UNSW AUSTRALIA

Security Design Principles

- Saltzer & Schroeder [SOSP '73, CACM '74]
  - **Economy of mechanism** – KISS
  - **Fail-safe defaults** – as in 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

  UNSW AUSTRALIA

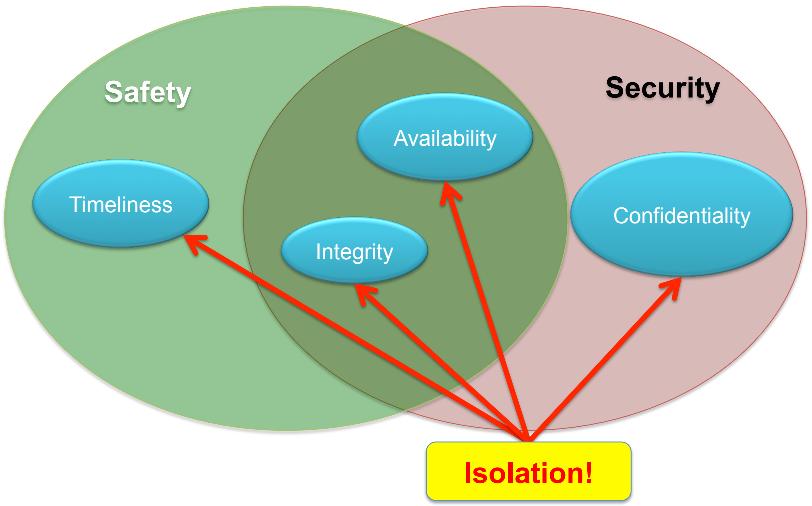
Common OS Security Mechanisms

- 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 Y
  - **Integrity**
    - X should not be tampered with by Y
  - **Availability**
    - X should not be made unavailable to Z by Y

Security vs Safety



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 assumptions
- Good assumptions:
  - clearly identified
  - verifiable

Risk Management

- Comes down to **risk management**
  - At the heart of all security
  - Assumptions: risks we are willing to tolerate
- Other risks:
  - we mitigate (using security mechanisms)
  - or transfer (e.g. by buying insurance)
- Security policy should distinguish which is appropriate for each risk
  - Based on a thorough **risk assessment**

Trust

- Systems always have **trusted** entites
  - whose misbehaviour can cause insecurity
  - hardware, OS, sysadmin ...
- **Trusted Computing Base** (TCB):
  - the set of all such entities
- Secure systems require **trustworthy** TCBs
  - achieved through assurance and verification
  - shows that the TCB is unlikely to misbehave
  - Minimising the TCB is key for ensuring correct behaviour

Assurance and Formal Verification

- **Assurance**:
  - systematic evaluation and testing
- **Formal verification**:
  - mathematical proof
- Together trying to establish correctness of:
  - the **design** of the mechanisms
  - and their **implementation**
- **Certification**: independent examination confirming that the assurance or verification was done right

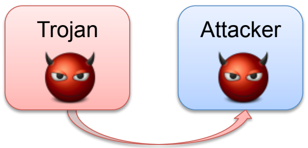
Covert Channels

- Information flow not controlled by security mechanisms
  - Confidentiality requires absence of all such
- **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

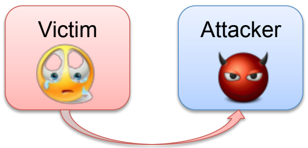
Covert Timing Channels

- Created by shared resource whose timing-related behaviour can be monitored
  - network bandwidth, CPU load ...
- Requires access to a time source
  - anything that allows processes to synchronise
  - Generally compare relative occurrence of two event sequences (clocks)
- 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.

Covert Channels vs Side Channels



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



- Attacker uses signal created by victim's innocent operations
- Much lower bandwidth

Summary: 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

  UNSW

Access Control

- **who** can access **what** in which ways
  - the "who" are called **subjects**
    - 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

  UNSW

AC Mechanisms and Policies

- AC Policy
  - Specifies allowed accesses
  - And how these can change over time
- AC Mechanism
  - Implements the policy
- Certain mechanisms lend themselves to certain kinds of policies
  - Some policies cannot be expressed using your OS's mechanisms

  UNSW

Protection State

Access control matrix defines the **protection state** at particular time

	Obj1	**Obj2**	**Obj3**	**Subj2**
Subj1	R	RW		send
Subj2		RX		control
Subj3	RW		RWX own	recv

Note: All subjects are also objects!

  UNSW

Storing Protection State

- Not usually as access control matrix
  - too sparse, inefficient, dynamic
- Two obvious choices:
  - store individual columns with each object
    - defines the subjects that can access each object
    - each such column is called the object's **access control list**
  - store individual rows with each subject
    - defines the objects each subject can access aka subject's **protection domain**
    - each such row is called the subject's **capability list**

Access Control Lists (ACLs)

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

	Obj1
Subj1	R
Subj2	
Subj3	RW

Capabilities

- A **capability** [Dennis & Van Horn, 1966] is a capability list element

	Obj1	**Obj2**	**Obj3**	**Subj2**
Subj1	R	RW		send

  - **Names** an object to which the capability refers
  - **Confers** permissions over that object
- Capability is **prima facie authority** to perform an operation
  - System will perform operation iff appropriate capability is presented
- Less common in commercial systems
  - IBM System→38-AS/400→i-Series
  - KeyKOS (Visa transaction processing) [Bromberger et al, 1992]
  - More common in research: EROS, Cheri, seL4

Capabilities: Implementations

- Capabilities must be unforgeable
  - Traditionally protected by hardware (tagged memory)
  - Can be copied etc like data
- On conventional hardware, either:
  - Stored as ordinary user-level data, but unguessable due to sparseness
    - contains password or secure hash
  - Stored separately (in-kernel), referred to by user programs by index/ address
    - "partitioned" or "segregated" capabilities
    - like UNIX file descriptors
- Sparse capabilities can be leaked more easily
  - Huge amplification of covert channels!
- The only solution for most distributed systems

ACLs and Capabilities: Duals?

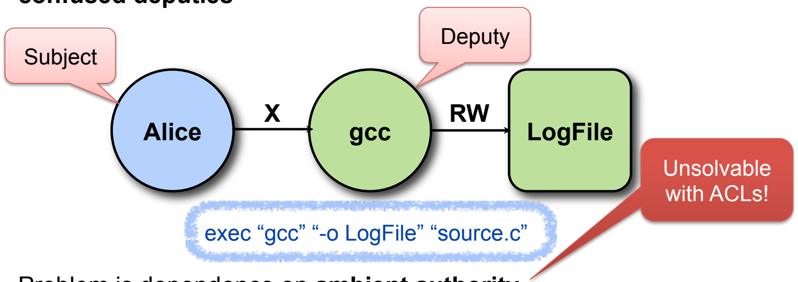
- In theory:
  - Dual representations of access control matrix
- Practical differences:
  - Naming and namespaces
    - Ambient authority
    - Deputies
  - Evolution of protection state
  - Forking
  - Auditing of protection state

Duals: Naming and Namespaces

- ACLs:
  - objects referenced by **name**
    - 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

Duals: Confused Deputies

- ACLs: separation of object naming and permission can lead to **confused deputies**



- Problem is dependence on **ambient authority**
  - Deputy uses its own authority when performing action on behalf of client
- Capabilities are both names and permissions
  - You can't name something without having permission to it
  - Presentation is normally explicit (not ambient)

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 capabilities
  - 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 capability to B that carries appropriate permissions
    - A can delegate X to B only if it has **grant** authority on X

Duals: Forking

- 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
  - Bad for 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

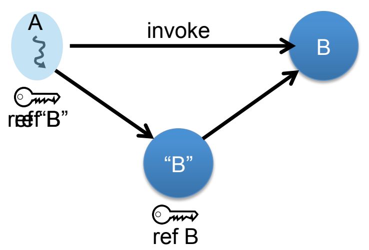
- Who has permission to access a particular object (right now)?
  - ACLs: Just look at the ACL
  - Caps: hard to determine with sparse or tagged caps, or for partitioned
- What objects a can particular subject access (right now)?
  - Capabilities: Just look at its capabilities
  - ACLs: may be impossible to determine without full scan
- "Who can access my stuff?" vs. "How much damage can X do?"

Interposing Object Access

Caps are opaque object references (pure names)

- Holder cannot tell which object a cap references nor the authority
- Supports transparent interposition (virtualisation)

Nice student project!



Usage:

- API virtualisation
- Security monitor
  - Security policy enforcement
  - Info flow tracing
  - Packet filtering…
- Secure logging
- Debugging
- Lazy object creation
  - Initial cap to constructor
  - Replace by proper object cap

Duals: Saltzer & Schroeder Principles

Security Principle	ACLs	Capabilities
Economy of Mechanism	Dubious	Yes!
Fail-safe defaults	Generally not	Yes!
Complete mediation	Yes (if properly done)	Yes (if properly done)
Open design	Neutral	Neutral
Separation of privilege	No	Doable
Least privilege	No	Yes
Least common mechanism	No	Yes
Psychological acceptability	Neutral	Neutral

Mandatory vs. Discretionary AC

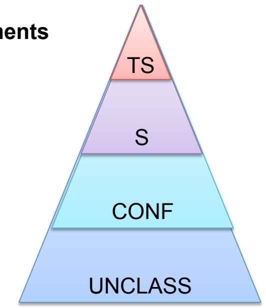
- Discretionary Access Control:
  - Users can make access control decisions
    - delegate their access to other users etc.
- Mandatory Access Control (MAC):
  - enforcement of administrator-defined policy
  - users cannot make access control decisions (except those allowed by mandatory policy)
  - can prevent untrusted applications running with user's privileges from causing damage

MAC

- Common in areas with global security requirements
  - e.g. national security classifications
- Less useful for general-purpose settings:
  - hard to support different kinds of policies
  - all policy changes must go through sysadmin
  - hard to dynamically delegate only specific rights required at runtime

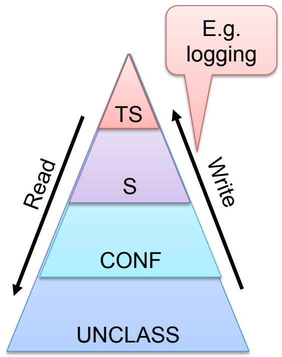
Bell-LaPadula [1966] (BLP) Model

- MAC Policy/Mechanism
  - Formalises National Security Classifications
- Every object assigned a **classification**
  - e.g. TS, S, C, U
  - may also have 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



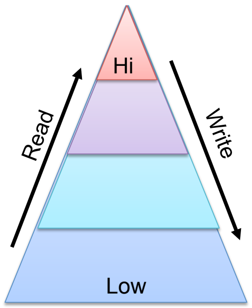
BLP: Rules

- **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



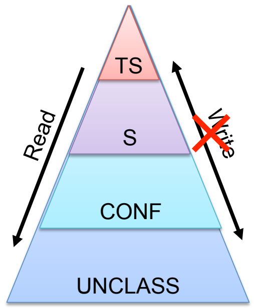
Biba Integrity Model

- 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

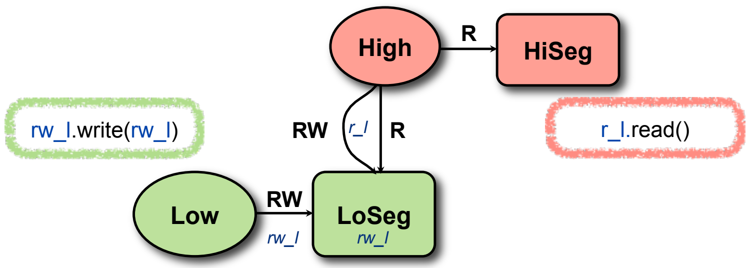
- BLP+Bibra allows no information flow across classes
  - Assume high-classified subject to treat low-integrity info responsibly
  - Allow read-down
- **Strong *-Property** ("matching writes only"):
  - s can write o iff clearance(s) = class(o)
  - Eg for logging, high reads low data and logs



Boebert's Attack

"On the Inability of an Unmodified Capability Machine to Enforce the *-Property" [Boebert 1984]

- Shows an attack on capability systems that violates the *-property
  - Low passes cap to write buffer to High, which can then write down
  - Where caps and data are indistinguishable (sparse, tagged)
  - Does not work against **partitioned** capability systems



Boebert's Attack: Lessons

- Not all mechanisms can support all policies
- Many policies treat data- and access-propagation differently
  - Eg explicit **grant** capability (Take-grant model)
  - Cannot be expressed using sparse capability systems
- This does **not** mean that capability systems and MAC are incompatible in general

Decideability

- Boebert's attack highlights the need for **decideability** of safety in an AC system
- **Safety Problem:** given an initial protection state s, and a possible future protection state s', can s' be reached from s?
  - i.e. can an arbitrary (unwanted) access propagation occur?
- **Harrison, Ruzzo, Ullman [1975] (HRU):**
  - undecideable in general
  - equivalent to the halting problem

Decideable AC systems

- The safety problem for an AC system is **decideable** if we can always answer this question mechanically
- Most capability-based AC systems decideable:
  - instances of Lipton-Snyder **Take-Grant** access control model [1977]
  - Take-Grant is decideable in linear time
- Less clear 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
- Certain kinds of policies cannot be enforced with certain kinds of mechanisms
  - e.g. *-property with sparse capabilities
- AC systems should be decideable
  - so we can reason about them

ACCESS CONTROL PRACTICE

Case Study: SELinux

- NSA-developed MAC for Linux
  - Based on **Flask** [Spencer & al., 1999]
- Designed to protect systems from buggy applications
  - Especially daemons and servers that have traditionally run with superuser privileges
- Adds a layer of MAC atop Linux's traditional DAC
  - Each access check must pass both the normal DAC checks and the new MAC ones
- Used widely in e.g. Enterprise linux

SELinux: Policy

- **Domain-Type Enforcement:**
  - Each process labelled with a **domain**
  - Each object labelled with a **type**
  - Central policy describes allowed accesses from domains to types
- Example:
  - `named` runs in `named_d` domain; `/sbin` labelled with `sbin_t` type
  - "allow named_d sbin_t:dir search"
  - Domain assignment for new processes on exec()
    - based on exec'ing domain and exec'd file type
    - "type_transition initrc_d squid_exec_t:process squid_d"
  - Type assignment to new files/directories
    - based on domain of creator process and type of parent directory
    - "`type_transition named_t var_run_t:sock_file named_var_run_t`"

SELinux

- Static fine-grained MAC
- Monolithic policy of high complexity
  - "The simpler targeted policy consists of more than 20,000 concatenated lines ... derived from ... thousands of lines of TE rules and file context settings, all interacting in very complex ways."
    - Red Hat Enterprise Linux 4: Red Hat SELinux Guide, Chapter 6. Tools for Manipulating and Analyzing SELinux
- Limited flexibility
  - What authority should we grant a text editor?
    - Needed authority determined only by user actions

Case Study: Capsicum

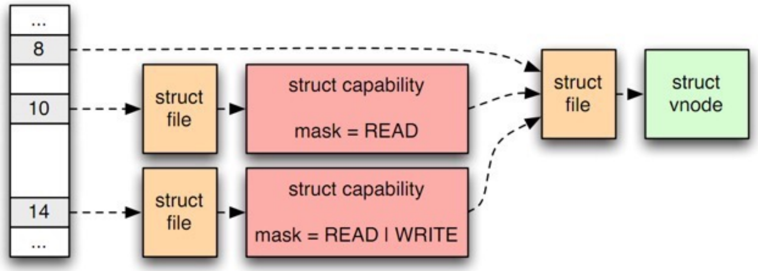
- "Practical Capabilities for UNIX" [Watson et al., 2010]
- Designed to support least privilege in conventional systems
  - without downsides of MAC
  - through delegation
- Merged into FreeBSD 9

Capsicum: Kernel

- Capsicum adds to the FreeBSD kernel:
  - Capabilities with fine-grained access rights for standard objects (files, processes etc.)
  - Capability Mode
    - Disallows access to global namespaces (e.g. filesystem etc.)
    - All accesses must go through capabilities
    - *at() system calls can resolve only names “underneath” the passed descriptor
    - Allows access to subsets of the filesystem by directory capabilities

FreeBSD Capsicum: Capabilities

- New file descriptor type
  - Wrap traditional file descriptors
  - Carry fine-grained access rights



FreeBSD Capsicum: Capabilities

- Capability passing as for file descriptors:
  - may be inherited across fork()
  - passed via UNIX domain sockets
- Created using cap_new()
  - From a raw file descriptor and a set of rights
  - Or an existing capability
    - New cap’s rights must be a subset
- Capabilities may refer to files, directories, processes, network sockets etc.

FreeBSD Capsicum: Capability Mode

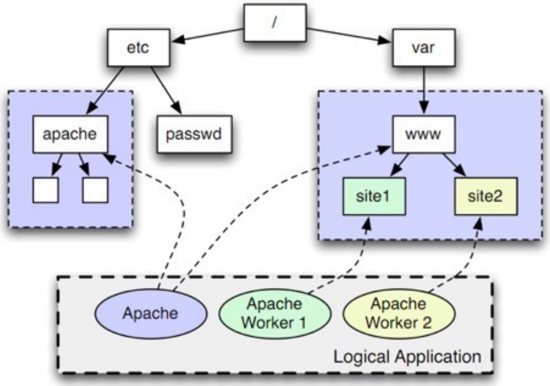
- Entered via new syscall: cap_enter()
  - Sets a flag that all child processes then inherit and can never be cleared once set
- Disallows access to all global namespaces:
  - Process ID (PID), file paths, protocol addresses (e.g. IP addrs), system clocks etc.
    - e.g. open() syscall disallowed (but openat() OK)
  - All accesses through delegated capabilities
    - Removes all ambient authority

FreeBSD Capsicum: *at() syscalls

- Allow lookups of paths relative to a given directory
  - specified by a directory file descriptor
  - e.g. `openat(rootdirfd,"somepath", O_RDONLY)`
- In capability mode, prevented from traversing any path above the given cap
  - e.g. `openat(dirfd,"../blah", flags)` disallowed
  - Ensures that directory caps do not confer authority to access their parents

FreeBSD Capsicum: Capability Mode

- Directory capabilities allow access to sub-parts of the filesystem namespace



FreeBSD Capsicum: Delegation

- A parent delegates to an app it invokes by:
  - fork()ing, obtaining a cap to the child
  - child drops or weakens unneeded caps, calls cap_enter(), then exec()s invoked binary
- Allows e.g. your shell to delegate sensibly to apps it invokes
  - Although apps need to be modified to do all accesses via capabilities
  - Provides an incremental path towards security

AC Mechanisms and Least Privilege

- Secure OS should support writing least-privilege applications
  - decomposing app into distinct components
  - each of which runs with least privilege
- Largely comes down to its AC system
  - some make this far more easy than others
- Example: web browser
  - handles lots of the user's sensitive info
  - but processes lots of untrusted input
  - input processing parts need to be sandboxed

Sandboxing Chromium [Watson et al., 2010]

	OS	Sandbox	LOC	FS	IPC	Net	Priv
DAC	Windows	DAC ACLs	22,350	⚠	⚠	✗	✓
	Linux	chroot()	600	✓	✗	✗	✗
MAC	OS X	Sandbox	560	✓	⚠	✓	✓
	Linux	SELinux	200	✓	⚠	✓	✗
Caps	Linux	seccomp	11,300	⚠	✓	✓	✓
	FreeBSD	Capsicum	100	✓	✓	✓	✓

USABLE SECURITY

Users and Security

- "The single biggest cause of network security breaches is not software bugs and unknown network vulnerabilities but user stupidity, according to a survey published by computer consultancy firm @Stake."
  – http://www.zdnetasia.com/staff-oblivious-to-computer-security-threats-21201228.htm
- "if [educating users] was going to work, it would have worked by now."
  – http://www.ranum.com/security/computer_security/editorials/dumb/

Security Advice

- Security advice:
  – e.g. check URLs / HTTPS certs, use strong passwords, don't write down passwords, etc.
- Is regularly rejected:
  – when it makes it impossible to get work done
    - why bosses share their passwords with their PAs
  – when there is some incentive to do so
    - why users give out their passwords for chocolate
  – when nobody ever sees any threat
    - why nobody checks HTTPS certificates
    - who here has ever faced a live MITM?

Security Advice Rejection

- Is often rational (Herley, NSPW 2009)
  - because it costs more to follow it than not to
    - advice imposes a cost on **everyone**
    - but only a **fraction** ever get attacked
    - so for most, there is not benefit
- Is because security is secondary concern
  - people get paid (only) for getting work done
- Writing good security advice is **hard**
  - this says more about poor system design than about the motivations of end-users
- Good example: forced regular password changes
  - Forces users to choose weak passwords ⇒ weakens security
  - Lost productivity due to change, forgotten passwords ⇒ high cost
  - Vulnerability is still months, hackers need minutes ⇒ no security gain

Classical security theatre

A brief digression...

- Has your bank ever reminded you not to forget your ATM card when withdrawing cash?

User Education

- Needed when the most secure way to use a system differs from the easiest
  - for rational users: “easiest” = “most profitable”
    - will be different for different people
- Is expensive
  - Cheaper to avoid need for it by careful design
- Not always possible to avoid:
  - when security and productivity goals conflict
  - e.g. need-to-know versus intelligence sharing post 9/11

Why Usable Security?

- Design Principle: Make the easiest way to use a system the most secure
  - c.f. safe defaults
- In general: exploit the user to make the system more, not less, secure
  - by aligning their incentives to produce behaviour that enhances security
  - requires good understanding of economics, human behaviour, psychology etc.
    - why these are now becoming hot topics in security research

Secure Interaction Design

- Users often behave “insecurely” because their actions cause effects different to what they expect
  - User types password into a phishing website
    - did not expect the website was fraudulent
  - User executes email attachment
    - did not expect the attachment to be dangerous
- General principle: secure systems must behave in accordance with user expectations

User Expectations

- To behave in accordance with user expectations:
  - Software must clearly convey consequences of any security choices presented to user
  - Software must clearly inform the user to keep accurate their mental model that informs their choices
- Why secure UIs require **trusted paths**
  - Essential security mechanism of a secure OS

Trusted Path

- Unspoofable I/O with the user
  - unspoofable output
    - so the user can believe what they see
  - unspoofable input
    - so the user knows what they say will be honoured
- Requires trustworthy I/O hardware
- For interactions via the OS, requires:
  - trustworthy drivers
  - trustworthy kernel

Secure Attention Key

- A trusted path for logging in
  - Ctrl-Alt-Del in Windows NT-based systems
  - Untrappable by applications, so unspoofable
  - Traps directly to kernel
  - Causes login prompt only to be displayed
- Requires user effort
  - So not optimal
  - But better than nothing

Hardware Trusted Paths

- For high-security situations, often cannot trust kernel or device drivers
- These use hardware-only trusted paths
  - Simple I/O hardware directly connected to security-critical device functions
    - e.g. pushbuttons (input) and LEDs (output)
  - bypasses OS
    - requires only that the hardware is trusted

Usable Security: Summary

- Design OS security mechanisms with real users in mind
  - mechanisms that fail when users behave normally are faulty, not the other way around
- Mechanisms must convey accurate information to users
  - so they can make informed security decisions
- Mechanisms should infer security decisions from normal user actions
  - granting authority according to least privilege

ASSURANCE AND VERIFICATION

Assurance: Substantiating Trust

- Specification
  - unambiguous description of desired behaviour
- System design
  - justification that it meets specification
    - by mathematical proof or compelling argument
- Implementation
  - justification that it implements the design
    - by proof, code inspection, rigorous testing
- Maintenance
  - justifies that system use meets assumptions

Common Criteria

- Common Criteria for IT Security Evaluation [ISO/IEC 15408, 99]
  - ISO standard, for general use
  - evaluates QA used to ensure systems meet their requirements
  - Developed out of the famous US DOD “Orange Book”: *Trusted Computer System Evaluation Criteria* [1985]
- **Target of Evaluation (TOE)** evaluated against **Security Target (ST)**
  - **ST:** statement of desired security properties based on **Protection Profiles**

Common Criteria: EALs

- 7 **Evaluated Assurance Levels**
  - **higher levels = more thorough evaluation**
    - **higher cost**
    - **not necessarily better security**

EAL 1–4 “not for use in hostile environments”

Level	Requirements	Specification	Design	Implementation
EAL1	not eval.	**Informal**	not eval.	not eval.
EAL2	not eval.	**Informal**	**Informal**	not eval.
EAL3	not eval.	**Informal**	**Informal**	not eval.
EAL4	not eval.	**Informal**	**Informal**	not eval.
EAL5	not eval.	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL6	**Formal**	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL7	**Formal**	**Formal**	**Formal**	**Informal**

Common Criteria Protection Profiles (PPs)

- Controlled Access PP (CAPP)
  - standard OS security, up to EAL3
- Single Level Operating System PP
  - superset of CAPP, up to EAL4+
- Labelled Security PP
  - MAC for COTS OSes
- Multi-Level Operating System PP
  - superset of CAPP, LSPP, up to EAL4+
- Separation Kernel Protection Profile (SKPP)
  - strict partitioning, for EAL6-7

COTS OS Certifications

- EAL3:
  - Mac OS X
- EAL4:
  - 2003: Windows 2000
  - 2005: SuSE Enterprise Linux
  - 2006: Solaris 10 (EAL4+)
    - against CAPP (an EAL3 PP!)
  - 2007: Red Hat Linux (EAL4+)
- EAL6
  - Green Hills INTEGRITY-178B (EAL6+)
    - against SKPP
    - relatively simple hardware platform in TOE

Get regularly hacked!

SKPP on Commodity Hardware

- SKPP:
  - OS provides only separation
- One Box One Wire (OB1) Project
  - Use INTEGRITY-178B to isolate VMs on commodity desktop hardware
  - Leverage existing INTEGRITY certification
    - by "porting" it to commodity platform
  - Conclusion [NSA, March 2010]:
    - SKPP validation for commodity hardware platforms infeasible due to their complexity
    - SKPP has limited relevance for these platforms
  - NSA subsequently dis-endorsed SKPP

Common Criteria Limitations

- Very expensive
  - rule of thumb: EAL6+ costs $1K/LOC
- Too much focus on development process
  - rather than the product that was delivered
- Lower EALs of little practical use for OSes
  - c.f. COTS OS EAL4 certifications
- Commercial Licensed Evaluation Facilities licenses rarely revoked
  - Leads to potential "race to the bottom" (Anderson & Fuloria, 2009)

Formal Verification

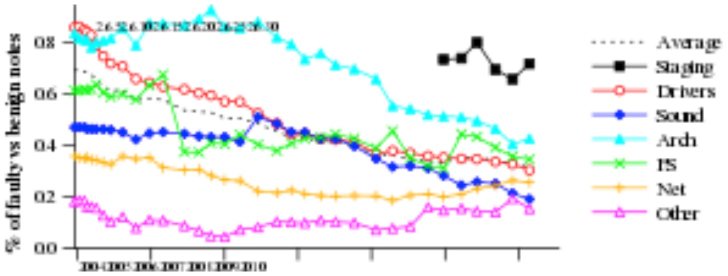
- Based on mathematical model of system
- Two approaches:
  - Automated techniques based on model checking / abstract interpretation
  - Theorem proving (manual or partially automated)

Automatic Analyses

- Algorithms that analyse code to detect certain kinds of defects
  - Usually static analysis
- Cannot generally "prove" code is correct
  - Only certain properties
  - False positives
  - False negatives
- Can be sound: guaranteed to detect all potential bugs of a kind
  - No false negatives
- Relatively cheap, often highly scalable (but then typically not sound)
  - Tradeoff between completeness and cost

Static Analysis and Linux: A Sad Story

- Static analysis of Linux source [Chou & al, 2001]
  - Found high density of bugs, especially in device drivers
- Re-analysis 10 years later [Palix & al, 2011]
  - Density of bugs detectable by static analysis had not dropped a lot!



Theorem Proving

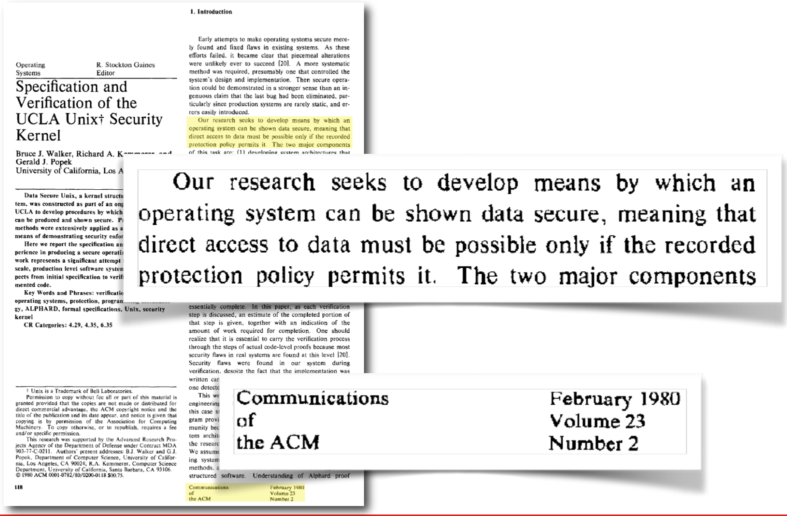
- State desired properties as a theorem in a mathematical logic
- Proof:
  - Model satisfies security properties
    - Required by CC EAL5-7
  - The code implements the model
    - Not required by any CC EAL (informal argument for EAL7)
- Example: seL4 microkernel
  - 2009: proof that code implements model
  - 2011: proof that model enforces integrity
  - 2013: proof that model enforces confidentiality
  - 2013: proof that binary is correct translation of C code

Formal Verification Limitations

- Proofs are expensive
  - e.g. seL4 took ~12 py for ~10,000 LOC
  - … plus a lot of re-usable effort and learning
  - **But:**
    - Factor 2–3 less expensive than Integrity EAL6+ certification
    - Factor 2–3 more expensive than traditional low-assurance code
- Proofs rest on assumptions
  - assume correct everything you don't model
    - e.g. details of hardware platform, etc.
  - difficult to assume that e.g. modern x86 platform is bug free!
  - full proofs best suited for systems that run on simple hardware platform
    - e.g. embedded systems
    - otherwise they're not yet worth the high cost

SEL4 AND SECURITY ASSURANCE

A 30-Year Dream


Operating Systems — R. Stockton Gaines, Editor

Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek
University of California, Los Angeles

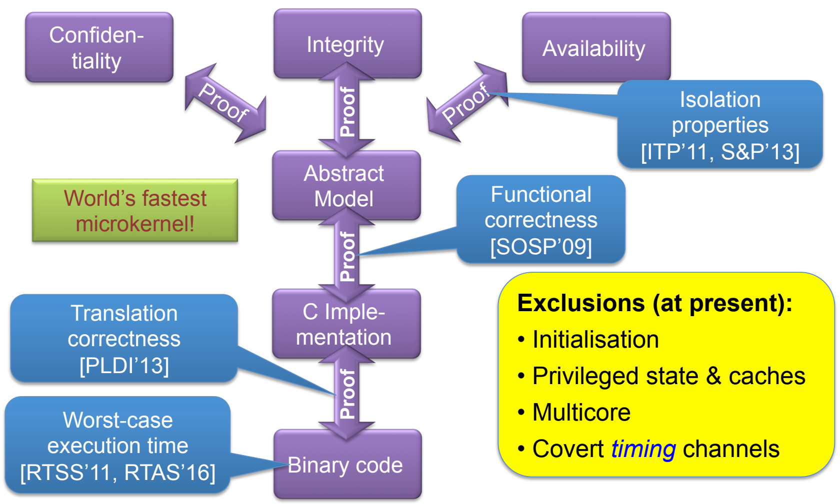
Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components

Communications of the ACM — February 1980, Volume 23, Number 2

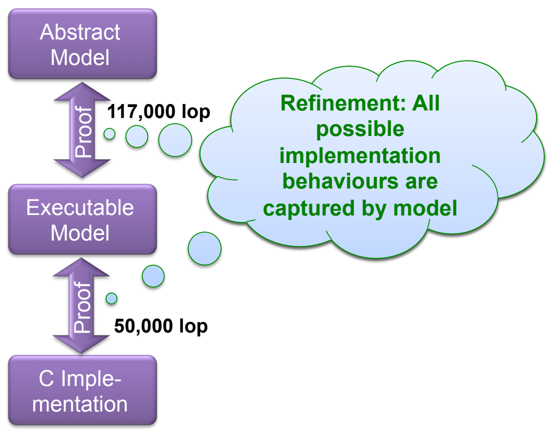
seL4 Assurance

Common Criteria	EAL4	EAL5	EAL6	EAL7	seL4
Requirements	Informal	Formal	Formal	Formal	Formal Proof
Functional Spec	Informal	Semiformal	Semiformal	Formal	Formal Proof
High-Level Design	Informal	Semiformal	Semiformal	Formal	Formal Proof
Low-Level Design	Informal	Informal	Semiformal	Semiformal	Formal Proof
Code	Informal	Informal	Informal	Informal	Formal

seL4 Provable Security Enforcement



Proving Functional Correctness



Proving Functional Correctness

```
constdefs
  schedule :: "unit s_monad"
  "schedule ≡ do
     threads ← allActiveTCBs;
     thread ← select threads;
     do_machine_op flushCaches OR return ();
     modify (λs. s ⦇ cur_thread := thread ⦈)
   od"
```

```
schedule :: Kernel ()
schedule = do
ad
curThread
meSlice curThread
ime == 0) chooseThread
```

```
void
setPriority(tcb_t *tptr, prio_t prio) {
    prio_t oldprio;

    if(thread_state_get_tcbQueued(tptr->tcbState)) {
        oldprio = tptr->tcbPriority;
        ksReadyQueues[oldprio] = tcbSchedDequeue(tptr, ksReadyQueues[
        if(isRunnable(tptr)) {
            ksReadyQueues[prio] = tcbSchedEnqueue(tptr, ksReadyQueues
        }
        else {
            thread_state_ptr_set_tcbQueued(&tptr->tcbState, false);
        }
    }

    tptr->tcbPriority = prio;
}

void
yieldTo(tcb_t *target) {
    target->tcbTimeSlice += ksCurThread->tcbTimeSlice;
```

  UNSW AUSTRALIA

MIT Technology Review

MIT Technology Review

10 BREAKTHROUGH TECHNOLOGIES 2011

Share

Crash-Proof Code

Making critical software safer

7 comments

WILLIAM BULKELEY

May/June 2011