UMSSIA

DAY II: THE QUEST FOR CONTROL…

OPERATING SYSTEMS

Typical OS classes focus on the **resource allocation** function of the OS.

Another important function of the OS is **resource protection**:

- Preventing errors in one process from effecting another’s
- Protecting the system’s resources from misuse.

If software security is about eliminating back doors, OS security is about closing the front door!

OS SECURITY ISSUES

OS Security is essentially concerned with three problems:

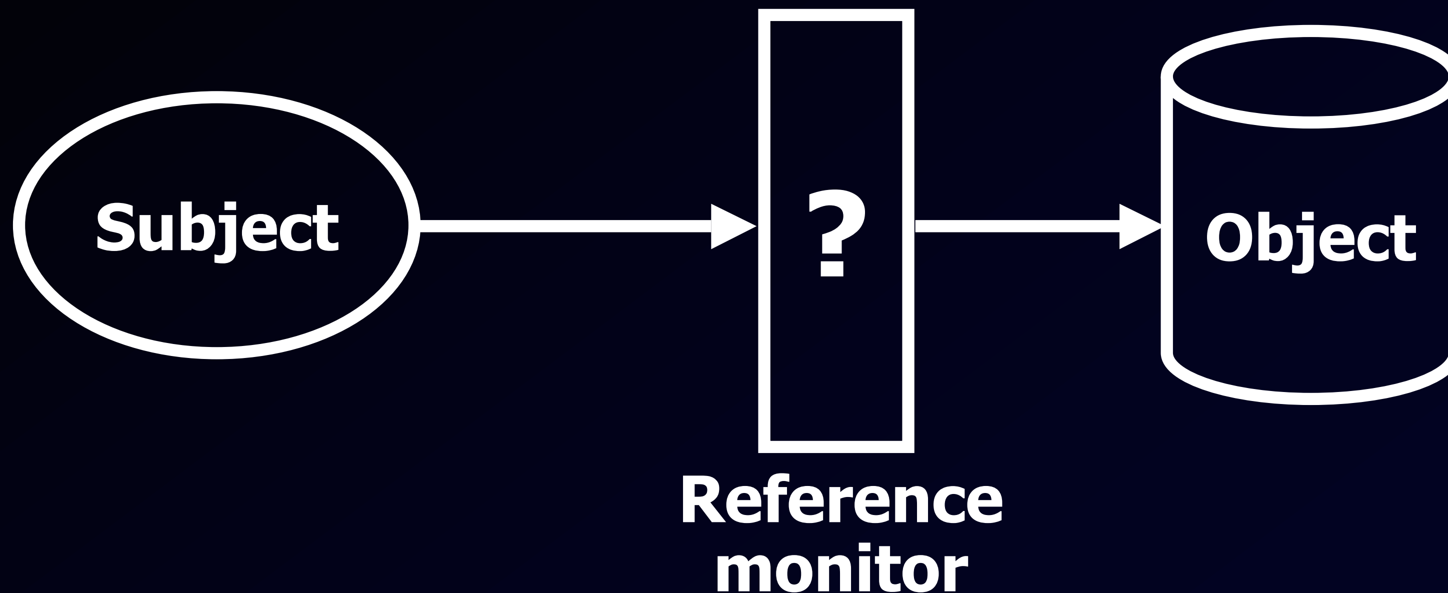
Access control is about deciding whether a program can access a resource.

Protection is the task of enforcing these decisions: ensuring a process does not access resources improperly.

Isolation is the separation of processes' resources from other processes.

ACCESS CONTROL

The OS **mediates** access requests between **subjects** and **objects**.



This mediation should (ideally) be impossible to avoid or circumvent.

TERMINOLOGY

Subjects make access requests on objects.

Subjects are the ones doing things in the system, like users, processes, and programs.

Objects are system resources, like memory, data structures, instructions, code, programs, files, sockets, devices, etc…

The type of access determines what to do to the object, for example execute, read, write, allocate, insert, append, list, lock, administer, delete, or transfer

ACCESS CONTROL **MATRIX**

Objects

Subjects

	Obj 1	Obj 2	Obj 3	…	Obj n
Subj 1	rwl	rwlx	-	-	l
Subj 2	rwl	rlx	rwl	-	-
Subj 3	-	-	-	rl	r
⋮				⋱	
Subj m	rl	lw	rl	rw	r

REPRESENTATIONS

An access control matrix can be represented internally in different ways:

	O1	O2	…
S1	rwl	wl	-
S2	ida	wlk	-
S3	-	-	rl
…			
Sm	rwlx	wi	w

Access Control Lists (ACLs) store the columns with the objects

Capability lists store the rows with the subjects

Role-based systems group rights according to the “role” of a subject.

ACCESS CONTROL **LISTS**

The ACL for an object lists the access rights of each subject (usually users).

To check a request, look in the object's ACL.

ACLs are used by most OSes and network file systems, e.g. NT, Unix, and AFS.

ACL **PROBLEMS**

To be secure, the OS must authenticate that the user is who (s)he claims to be.

To revoke a user's access, we must check every object in the system.

There is often no good way to restrict a process to a subset of the user's rights.

CAPABILITIES I

Capabilities store the allowed list of object accesses with each subject.

When the subject requests access to object O, it must provide a "ticket" granting access to O.

These tickets are stored in an OS-protected table associated to each process.

No widely-used OS uses pure capabilities.

Some systems have "capability-like" features: e.g. Kerberos, NT, OLPC (BitFrost)

CAPABILITIES II

Some research OS projects use capabilities as object names.

Instead of typing “cat file”, type “cat {ticket}” where ticket is an unforgeable reference.

e.g. “allow read,list on inode 42 - OS”

Without a ticket there is no way to reference the object: hax0rz can’t name the object they want to access!

ACL **VS. CAPABILITY**

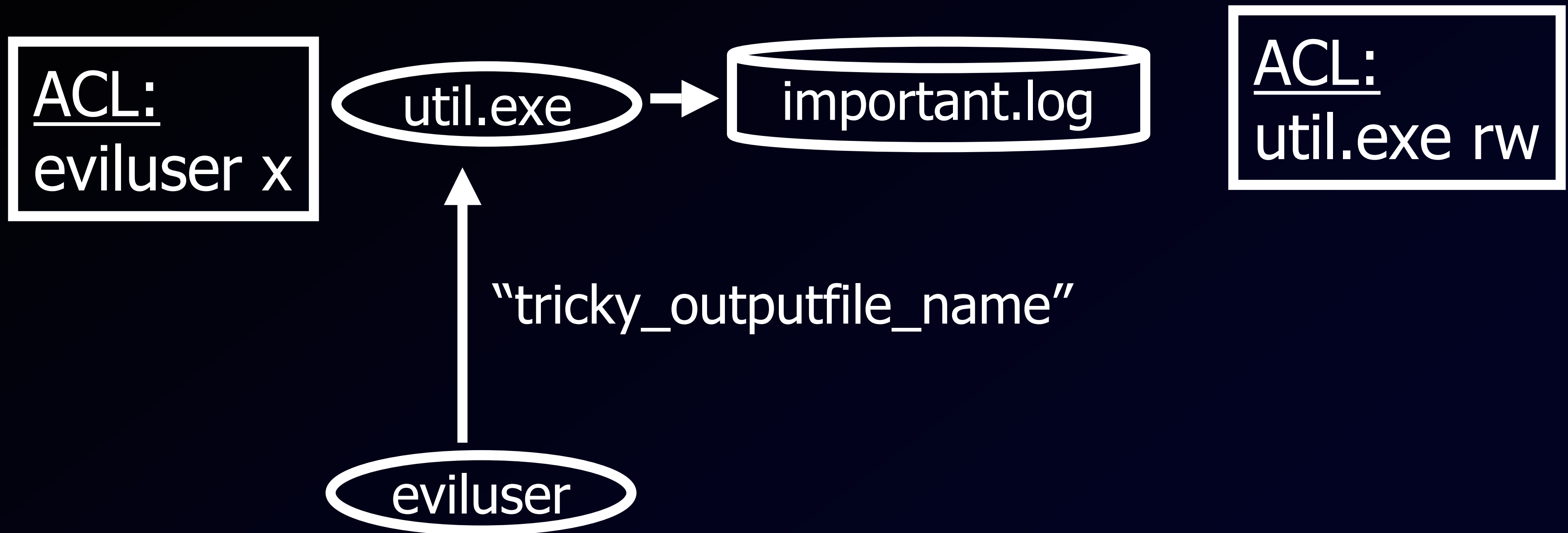
Capabilities do not require authentication: the OS just checks each ticket on access requests.

Capabilites can be passed, or delegated, from one process to another.

We can limit the privileges of a process, by removing unnecessary tickets from the table.

CONFUSED DEPUTIES

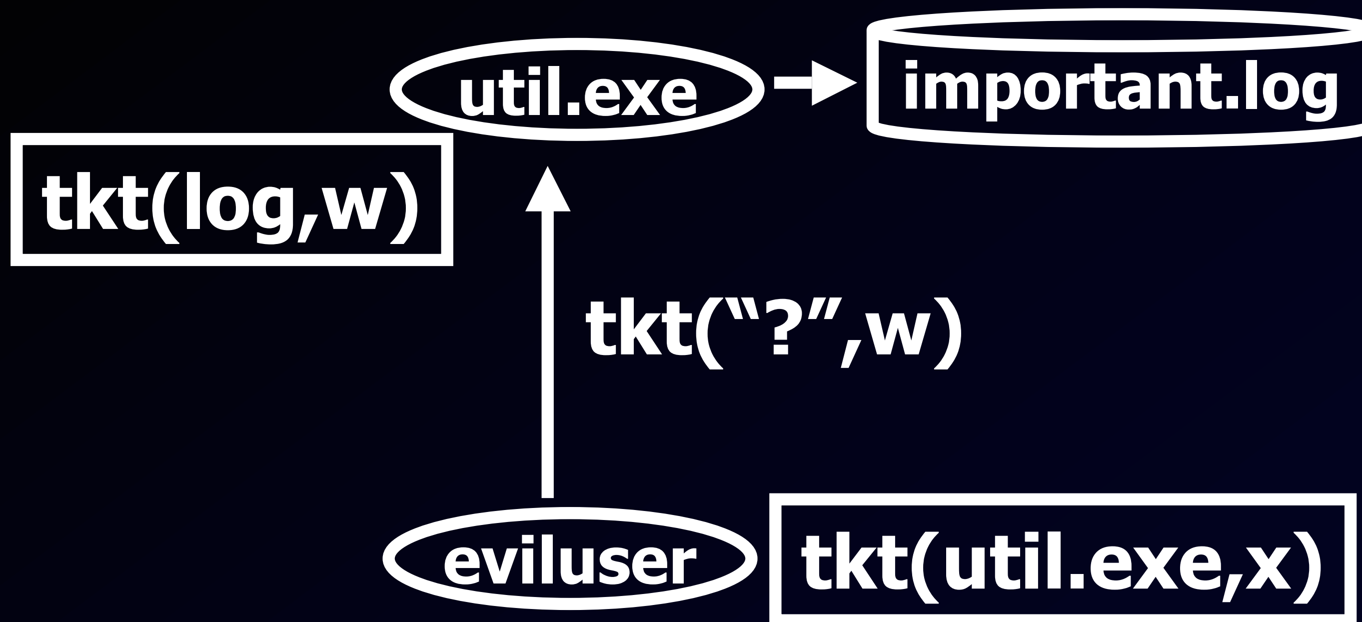
With ACL-style access control:



eviluser can write over important.log

LESS-CONFUSED DEPUTIES

With “pure” capability-style access control:



eviluser has no handle for important.log

REAL CONFUSED DEPUTIES

In some Unix systems, lpr/lpd runs as root. This made a big security hole:

1. **% lpr $file**
   copies $file onto lpr's tempfile
2. **% lpr –s $file**
   makes a symlink from tempfile to $file
3. **% ln –s /etc/passwd $file**
   now tempfile points to /etc/passwd
4. **% lpr mypasswd**
   copies mypasswd to tempfile = /etc/passwd!

PROBLEMS WITH CAPABILITIES

Suppose we revoke Alice's access to some resource. Should we revoke delegated capabilities? How?

To find out who can access an object, we need to list the capabilities of all subjects…

Usability can be a challenge when capabilities are used as object references.

ROLES

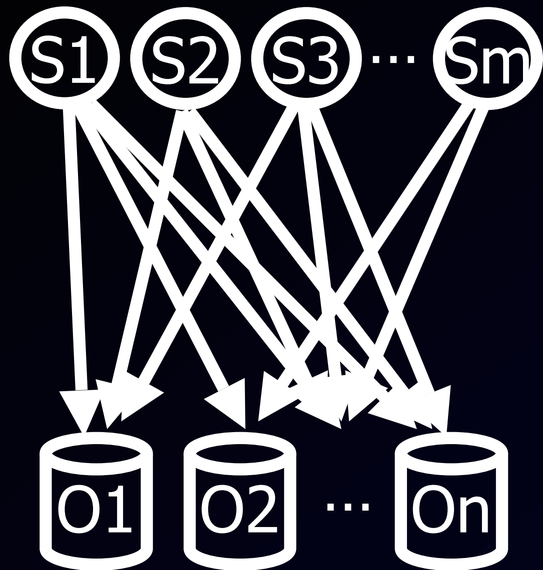
In many systems, subjects play one or more **roles**. This role includes access to objects.

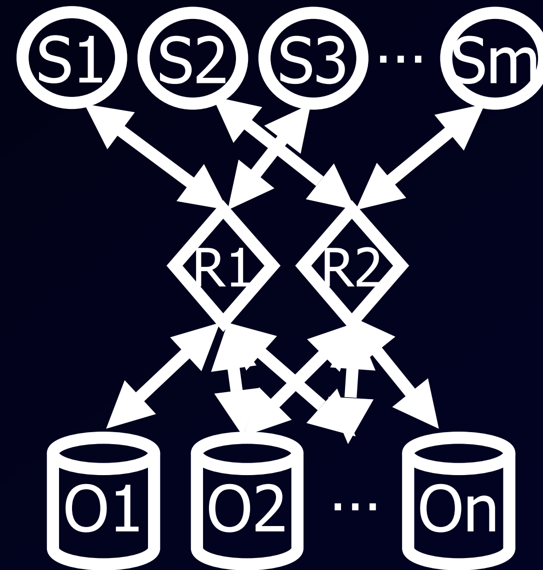
Users log in as their role, and can access the objects needed for that role.

For example a hospital might have roles like: Doctor, Nurse, Pharmacist, Receptionist, Lawyer, Accountant, Executive…

Because there are fewer roles than subjects, the access control matrix is stored efficiently.

ROLES





HARDWARE ACCESS CONTROL

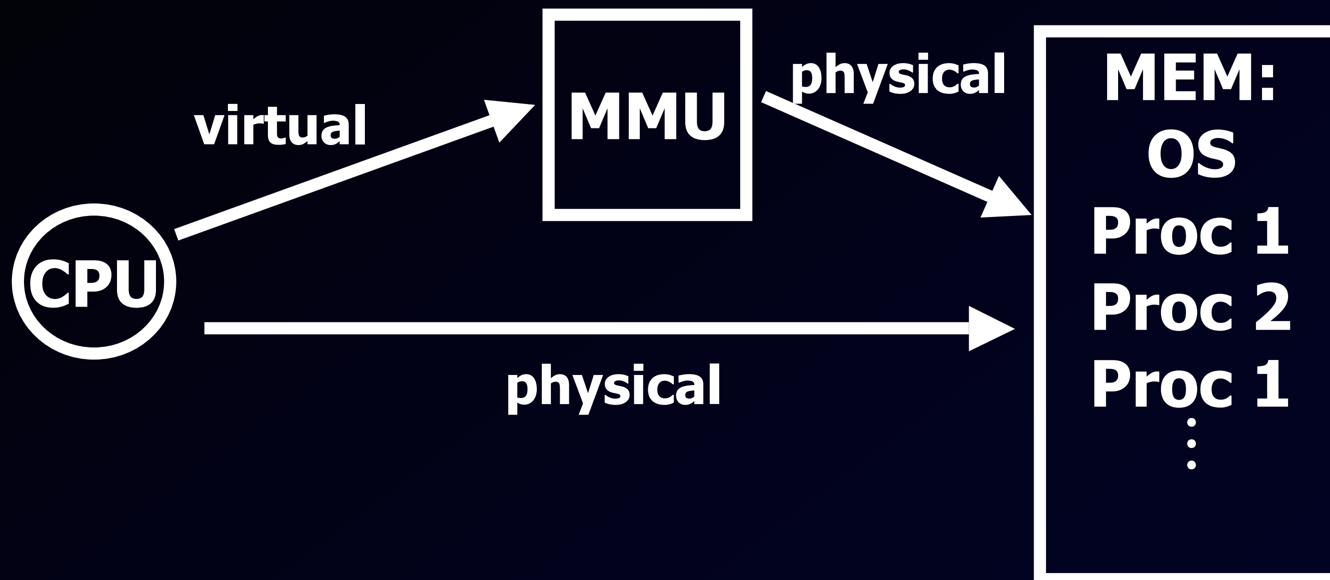
Access control, isolation, and protection are all implemented by the OS using hardware access control primitives.

Typically, hardware access control is “low granularity:” address translation + rings.

If there is no hardware support, we have two options: no access control or virtual hardware.

ADDRESS TRANSLATION

Processes access memory using **virtual addresses** that are translated into **physical addresses** by hardware:



The OS manages this translation so that each process sees only its own physical addresses.

RINGS

On most processors, user programs are prevented from changing translation or using direct access commands by using two or more "protection rings."

When the processor is operating in ring n, it can access memory and instructions in any ring m ¸ n.

USING RINGS FOR ACCESS CONTROL

Processors have a protection table listing a ring number for each memory segment, and a status register containing the current ring.

If user processes run in a higher ring, they cannot access OS data, like translation tables.

This separation is translated into access control by calling OS functions to access resources.

INWARD ACCESS

The OS creates an interrupt table to handle access requests, processor allocation, etc.

Each event causes the appropriate handler to be invoked, with appropriate privileges.

Typical events are TRAP instructions, timers, bus input, etc.

When the handler is done it is responsible for resetting the current ring level and MMU table.

USING RINGS: **UNIX**

Standard UNIX implementations use 2 rings:

The Kernel, system tables, and devices run in ring 0, or "supervisor mode."

Other processes and memory run in ring N, or "user mode."

System calls use an exception to go to ring 0, check access, allocate resources, and return to user mode.

UNIX USER SECURITY

A **user** in Unix is defined by a 16-bit **UID**, and is mapped to a textual name by **/etc/passwd**.

A **group** is defined by a 16-bit **GID**, defined by **/etc/groups**; users belong to 1 or more groups.

The command **su** is used to change UIDs; **newgrp** is used to change GIDs.

UNIX SUPERUSER

In UNIX, the special UID 0 is the superuser, or root.

Root can access anything on the system:

- Change any password
- Change permissions on any file
- Read, write, delete any file
- Create/access any device
- Change read-only filesystem to read/write
- Read /dev/mem and /proc

UNIX FILE SECURITY

- Each file has one owner and one group
- Each file has a permissions field:
  - Read, write and execute, by owner, group, and other; setuid, setgid, and sticky.
- Permissions are represented as vectors (- rwxr-xr--) or 4 octal digits: 0754
- The permissions field is set by the file's owner.
- The owner field can only be changed by root.

DIRECTORIES

- What do permissions mean for directories?
  - r: list contents of directory
  - x: access files in directory
  - w: create, rename, or remove files in dir
  - setgid: new files in directory owned by same group as . (Sys V)
  - setuid: new files in . owned by owner of . (BSD)
  - sticky: mv file / rm file only works for owner of file or dir

ACCESS RESOLUTION

- **Suppose file F has permission 177, can the owner read it?**
- **The algorithm for mediating access is:**
- **if uid = owner then *owner* permission**
  else if gid in group then *group* permission
  else *other* permission
- **So the owner and group of a file can have less privilege on a file than others…**

PROCESS UIDs

Each process has three user IDs:

- The Real User ID is: (RUID)
  - inherited from the parent process
  - used to determine which user started the process
- The Effective User ID: (EUID)
  - determines the permissions for the process
- The Saved User ID: (SUID)
  - is set after the EUID is changed, to allow a process to “go back” to the old EUID.

Similarly, each process has real and effective GIDs.

In Linux, each process has a “filesystem” UID and GID. These are used in place of EUID, EGID for file access.

UID MANAGEMENT

Fork sets a new process' UIDS to match the Parent process.

If file F has setuid turned on, **exec**(F) will set the process EUID to the UID of F's owner.

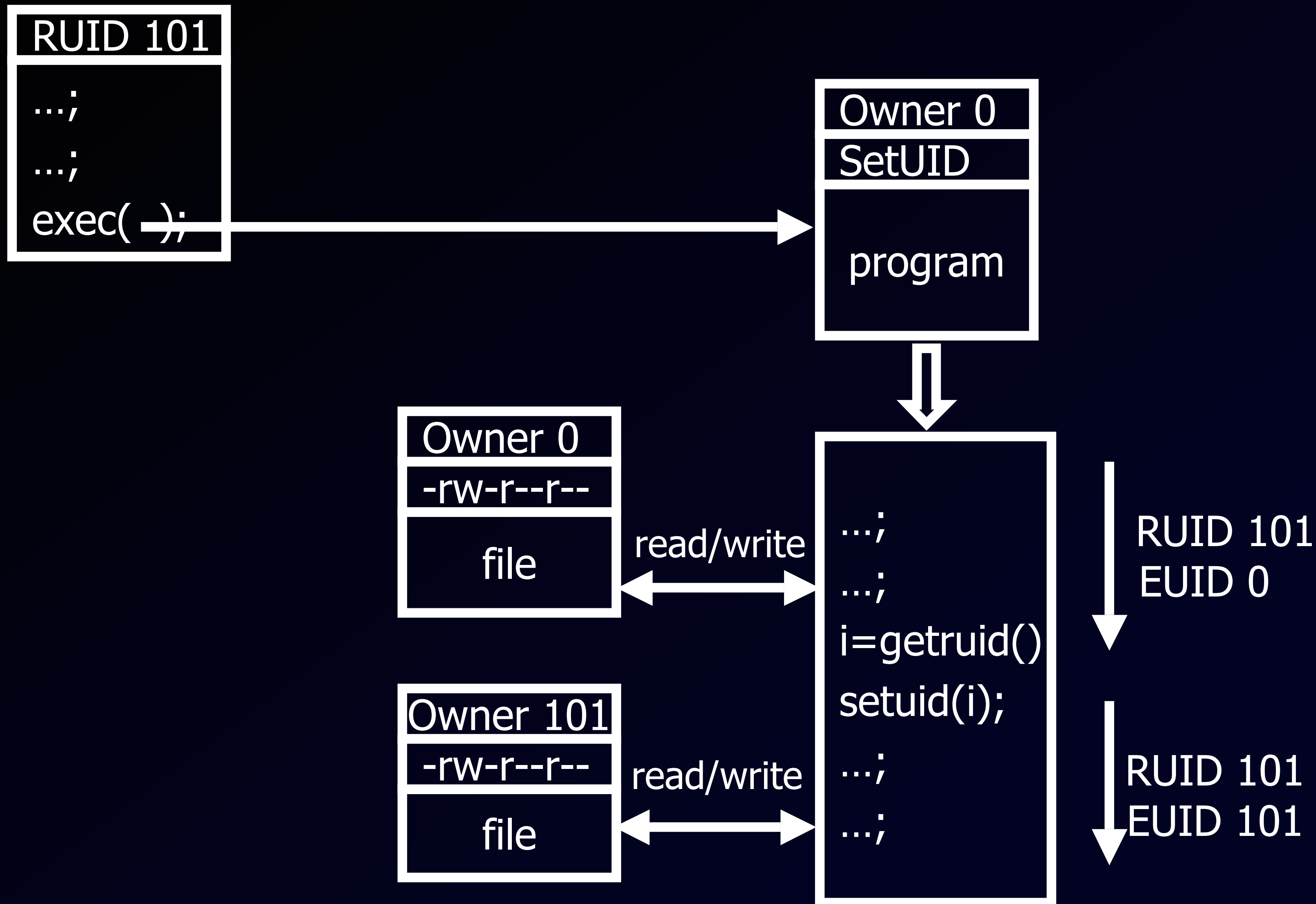
The OS provides calls to manipulate the process UID, e.g. seteuid(newid) can set EUID to:

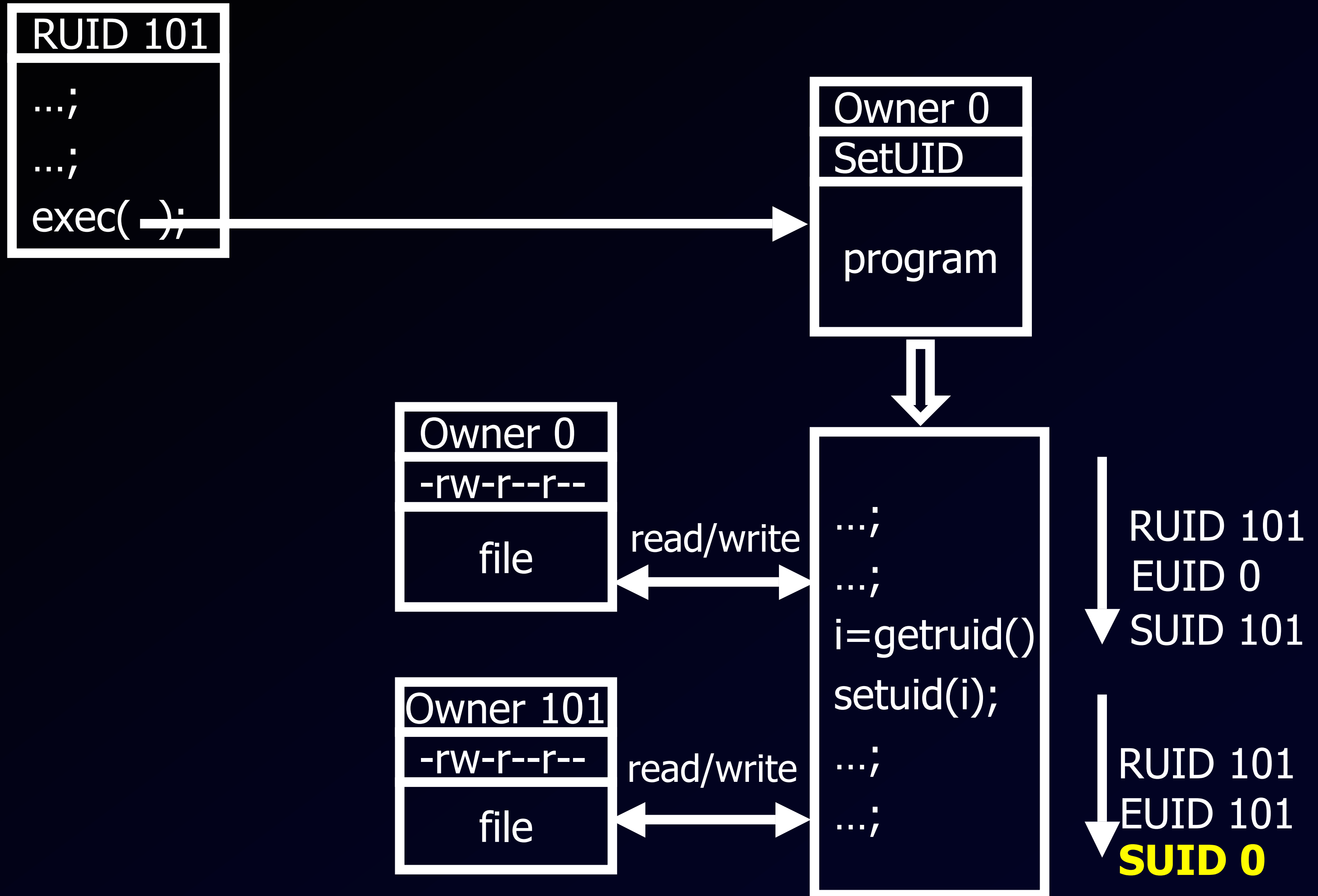
- SUID or RUID, at any time.
- Any ID, if EUID=0

If a system supports setresuid, it should be used to reliably drop privileges.

http://www.csl.sri.com/users/ddean/papers/usenix02.pdf

SETUID EXAMPLE




SUID PROBLEM
RUID 101
...;
...;
exec( );
Owner 0
SetUID
program
Owner 0
-rw-r--r--
file
read/write
Owner 101
-rw-r--r--
file
read/write
...;
...;
i=getruid()
setuid(i);
...;
...;
RUID 101
EUID 0
SUID 101
RUID 101
EUID 101
SUID 0

LEAST PRIVILEGE

- Unix provides some support for sandboxing with the chroot command:
  - chroot /home/jail/ dangerous-cmd
  - Runs dangerous-cmd with / = /home/jail.
  - Or syscall: chroot("/home/jail");
  - No way to see "real" /etc/passwd, /bin, etc.
  - Requires setup: /home/jail/lib should have all shared libraries, /home/jail/bin/sh should exist; config files in /home/jail/etc/…
- Chroot is not infallible – there are several options for "breaking out" of a chroot.

TRUSTWORTHY OS

- A principal is trusted if it can cause system security to fail.
- Thus any OS is trusted on the system it controls.
- A system is trustworthy if our trust in it is well-placed.
- How do we justify trusting an OS?

TRUSTWORTHY OS FEATURES

- Extra security features
  - access control, authentication
- More secure implementation
  - Secure design principles, coding practices
  - Minimal Trusted Computing Base (TCB)
  - Maintenance: patch in place, etc…
- Higher Assurance
  - Code audit
  - Penetration Testing
  - Formal Verification
  - Certification

DAC VS MAC

(Discretionary vs. Mandatory Access Control)

DAC

- Controlled by owner
- Users are trusted
- Decisions based on object ownership and uid
- Impossible to control information flow

MAC

- Object owner *may* have some control
- Only trust admins
- Objects and tasks themselves have ids
- Makes information flow control possible

EXAMPLE

SELinux implements a MAC mechanism called "Type Enforcement:"

- Every subject and object has a type.
- Every type has a set of operations.
- Each type may allow other types to access its operations.
- Operations may transition subject and object types.

The "SELinux Example Policy" enforces kernel and application integrity.

MULTILEVEL SECURITY

Multilevel Security (MLS) is based on US military-style security policy.

Each subject has a clearance level

Each object has a classification level

The security goal is to prevent leakage of classified information to uncleared subjects

MILITARY CLASSIFICATION

A classification consists of a rank along with a set of zero or more compartments.

We say S is cleared to access O if

- **Rank(S) ¸ Rank(O)**
- **Comp(S) ¶ Comp(O)**

When this happens, the classification of S dominates the classification of O.

EXAMPLE

Suppose we have the following ranks:

Unclassified < Confidential < Secret < Top Secret

And compartments: {EUR,ASIA,MAGIC}

If General Patton has clearance (TS, {EUR,MAGIC}), can he read a report classified (S, {EUR, ASIA})?

What classifications can General Patton read?

COMMERCIAL MLS EXAMPLES

- **A software company might have sensitivity levels like public, private, internal, and compartments like product_1,…, product_n, accounting, personnel,…**
- **Here, Developers on one product might not "need to know" about others. (From a security perspective)**
- **Managers may have access to some info that employees do not**

BELL-LAPADULA (BLP)

Bell and LaPadula formalized the MLS security policy that "subjects cannot read information above their clearance."

They used this to prove that if OS routines follow the policy, it will never be violated.

The basic idea is that we need two rules:

1. "no read up" and 2. "no write down."

BLP FORMALISM

BLP formally define a system by:

- **A set of subjects, *S*.**
- **A set of objects, *O*.**
- **A set of operations, *A* = {`read, write, execute, append,…`}**
- **A lattice of classifications, *L*.**
- **A set of states (who's accessing what right now)**
- **A set of transitions (syscalls that change the state)**

BLP **STATES**

Define the state of a system by:

- **The set *B* of current actions (s,o,a): *s* is accessing o with operation a.**
- **$f_c(s)$ = the current clearance of each subject s**
- **$f_S(s)$ = the maximum clearance of each s**
- **$f_O(o)$ = the classification of each object o**

BLP SECURITY

A state (B,f) is **secure** if it satisfies two properties:

- "ss-property": no read up, or

  8 (s,o,a) 2 B: (a = `r`)) $f_S(s)$ ¸ $f_O(o)$.

- "*-property": no write down, or

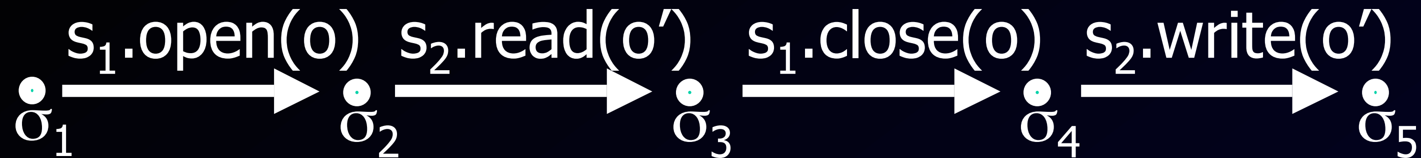
  8 (s,o,a) 2 B:

  (a 2 {`w,a`})) ($f_O(o)$¸ $f_C(s)$

  AND (8 (s,o',a') 2 B :

  (a' 2 {`r,w`})) $f_O(o')$ · $f_O(o)$))).

TRANSITIONS



A transition from state S_1 to S_2 is secure if S_1 and S_2 are secure.

The security of one system call can be checked independently of other calls.

BLP SECURITY THEOREM

Theorem. If all transitions are secure then
A system that starts in a secure state, will remain in a secure state.

Proof. Use induction on the length of transition sequences.

Notice that the proof does not depend on the specific BLP properties, so we can use it for other ss- and *-properties.

BIBA: MULTILEVEL INTEGRITY

In Biba's integrity policy, every subject and object is labeled with an integrity level.

For example, some objects might contain unreliable data, and we don't want to contaminate the system.

Biba has two security rules:

- **"Simple integrity": if s can modify o, then**
  $f_s(s)$ ¸ $f_o(o)$.
- **"Integrity *": if s can read o and s can write o' then**
  $f_o(o)$ ¸ $f_o(o')$.

BIBA POLICY EXAMPLES

- Embedded medical systems
- Airplanes
- Airlines
- Electric Company
- Dynamic: PERL tainting

CHINESE WALL POLICY

The Chinese Wall Policy models security in a consulting business. The security goal is to avoid a conflict of interest.

- **Example:**
  - **Alice works on the IPO for ACME Search**
  - **She also works on IPO for BETA Search**
  - **The goal of the CW Policy is to prevent Alice leaking info about ACME to BETA**
- **Informally, the Chinese Wall Policy rule states that there should be no information flow that causes COI.**

SYSTEM MODEL

The Chinese wall policy models a system state as a tuple (*C,O,S,Company, Conflicts, A, N*):

- ***C*** **is the set of Companies.**
- ***O*** **is the set of Objects.**
- ***S*** **is the set of Subjects.**
- ***Company*** **: *O* ! *C* maps objects to companies.**
- ***Conflicts*** **: *O* ! 2^C maps each object to the set of conflicting companies**
- ***A*** **½ *S* £ *O* £ {r,w} is the set of current accesses**
- ***N*** **is the history matrix, where $N_{s,o}$ = true iff subject s has accessed object o**

CW SECURITY PROPERTIES

- ss-property: A subject can only access an object if he has never accessed a conflicting object
- *-property: A subject can only write to an object if all objects he is reading are for the same company or publicly available.

COVERT CHANNELS

- Without being careful, there are many ways for a "Top Secret" process to leak info to a "Unclassified" process.
- One way to leak the file F:
  - P2 creates files $f_1 \ldots f_N$ where N is the number of bits in F.
  - P1 opens (and locks) the files f_i such that the ith bit of F is 1.
  - P2 tries to open all files. If open(f_i) fails, then F[i] = 1, else F[i]=0.
- This is an example of a covert channel.

COVERT CHANNEL MITIGATION

- “Information Flow” is undecidable.
- Can we at least detect covert communications ad hoc?
  - No! Provably undetectable any time the OS has uncertainty about the value of the channel.
- The best we can do is isolation (where feasible) and rate-limiting.
  - Standard methods: control input, equal resource allocation, disrupt timing accuracy,…

STRONG AUTHENTICATION

Trustworthy OSes typically supplement passwords with multi-factor authentication, requiring two or more of:

1. **Something you know (password, PIN)**
2. **Something you have (secure token)**
3. **Something you are (biometrics)**
4. **Something you *can do* (CAPTCHA)**
5. **Some*place* (4D) you are (spatiotemporal access control)**

PASSWORDS

User-chosen passwords are easy to guess: crackers recover 25-40% of passwords.

Randomly chosen or assigned passwords have many disadvantages:

- Low recall
- Often written down
- Password clutter

Some alternatives: first letters of phrases, “pronounceable” random passwords, password aging (watch for the “change back” attack) and cracker filters.

BIOMETRICS

… are physical characteristics that could be used to authenticate users. Some examples include:

- Signatures
- Face recognition
- Fingerprints
- Iris Scanning
- Voice Recognition
- Typing patterns
- DNA analysis
- Hand Geometry

Problems with biometric authentication include error rates, impersonation, privacy concerns, and device security.

ERROR RATES

- Many biometric systems are prone to error, e.g. failure of physical sensors, or the existence of "doppelgangers."
- The **False Positive Rate** is the probability that a randomly chosen Alice will be (incorrectly) accepted as Bob.
- The **False Negative Rate** or insult rate, is the probability that Alice will be (incorrectly) rejected as Alice.
- The **Equal Error Rate** is the FPR when the system's acceptance threshold is tuned so FPR=FNR.

BIOMETRICS

Impact of Artificial "Gummy" Fingers on Fingerprint Systems

Tsutomu Matsumoto
Hiroyuki Matsumoto
Koji Yamada
Satoshi Hoshino

Graduate School of Environment and Information Sciences
Yokohama National University
79-7 Tokiwadai, Hodogaya, Yokohama 240-8501, Japan
email: tsutomu@mlab.jks.ynu.ac.jp

ABSTRACT

Potential threats caused by something like real fingers, which are called fake or artificial fingers, should be crucial for authentication based on fingerprint systems. Security evaluation against attacks using such artificial fingers has been rarely disclosed. Only in patent literature, measures, such as "live and well" detection, against fake fingers have been proposed. However, the providers of fingerprint systems usually do not mention whether or not these measures are actually implemented in emerging fingerprint systems for PCs or smart cards or portable terminals, which are expected to enhance the grade of personal authentication necessary for digital transactions. As researchers who are pursuing secure systems, we would like to discuss attacks using artificial fingers and conduct experimental research to clarify the reality. This paper reports that gummy fingers, namely artificial fingers that are easily made of cheap and readily available gelatin, were accepted by extremely high rates by 11 particular fingerprint devices with optical or capacitive sensors. We have used the molds, which we made by pressing our live fingers against them or by processing fingerprint images from prints on glass surfaces, etc. We describe how to make the molds, and then show that the gummy fingers, which are made with these molds, can fool the fingerprint devices.

Keywords: biometrics, fingerprint, live and well detection, artificial finger, fingerprint image.

1. INTRODUCTION

Biometrics is utilized in individual authentication techniques which identify individuals, i.e., living bodies by checking physiological or behavioral characteristics, such as fingerprints, voice, dynamic signatures, etc. Biometric systems are said to be convenient because they need neither something to memorize such as passwords or something to carry about such as ID cards.[7, 11] In spite of that, a user of biometric systems would get into a dangerous situation when her/his biometric data are abused. That is to say, the user cannot frequently replace or change her/his biometric data to prevent the abuse because of limits of biometric data intrinsic to her/himself. Therefore, biometric systems must protect the electronic information for biometrics against abuse, and also prevent fake biometrics.

We have focused on fingerprint systems since they have become widespread as authentication terminals for PCs or smart cards or portable terminals. Some of fingerprint systems may positively utilize artificial fingers as substitutes in order to solve the problem that a legitimate user cannot access, for example, when s/he gets injured on her/his fingertip in an accident.[12] Some other cases include dry fingers, worn fingers, and other fingers with a low-quality fingerprint. However, the users of those systems would run a risk because artificial fingers can be stolen and used by other persons if the systems are utilized for a security application. Except for the above-mentioned cases, fingerprint systems generally must reject artificial fingers. In order to reject them, fingerprint systems should take measures to examine some other features intrinsic to live fingers than those of fingerprints. These measures are called "live and well detection[9, 15, 20, 23, 26]" and have been proposed mainly in patent literature.[13] Although a number of fingerprint systems have come into wide use, it is not clear whether or not these measures are actually implemented in commercial fingerprint systems. Moreover, as far as we know, security evaluation against attacks using fake fingers has been rarely disclosed. In connection, some researchers reported, in 1998, that four of the six fingerprint systems with optical devices accepted silicone fingers.[19] After that, some measures against silicone fingers may have been

TRUSTED PATH

A trusted path is an "unspoofable" method of communication between the OS and the user.

There are many possible implementations:

- **Using a unique key sequence, the hardware traps the "trusted path" key sequence and calls the OS. But how does the OS communicate with hardware?**

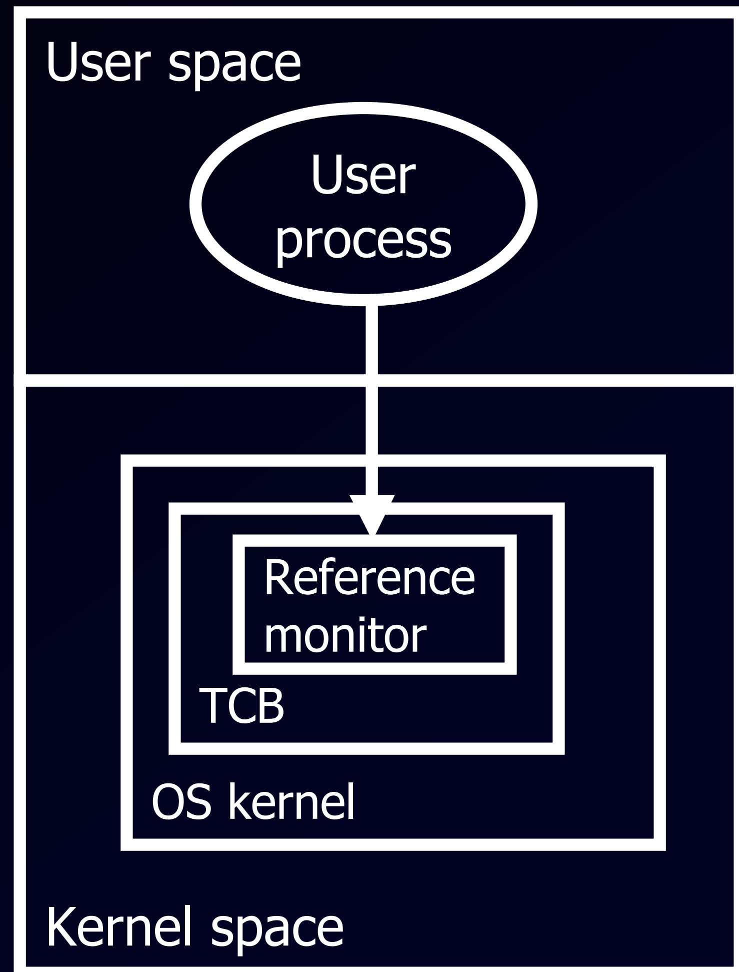
- **Using temporal separation, the user communicates with the security kernel before other applications start. Once applications start, how do we extend the trusted path?**

MINIMAL TCB

The **Trusted Computing Base** is the set of all hardware and software responsible for enforcing security rules

The Reference monitor is part of the TCB.

In a trustworthy OS, all system calls go through the reference monitor for security checking.



ASSURANCE

… is the process by which we gain confidence in the trustworthiness of an OS or system.

How do we gain assurance?

- **Testing** can only find errors, not show their absence.
- **Validation** is a "security process" combining requirements checking, design and code reviews, and module and system testing.
- **Formal verification** is the "gold standard" of assurance.

FORMAL ASSURANCE

- **Process, audit, and testing, cannot truly show something is "secure."**
- **To guarantee the OS has no holes, we need to write a proof that none exist.**
- **Informal proofs are prone to errors.**
- **Formal proofs can be verified by machines. They consist of a machine-readable list of every proof step, along with a justification.**
- **Short proofs can be done by machines; longer ones with help from humans.**

CERTIFICATIONS

How can we estimate the trustworthiness of a security product?

1) We can re-test the product to get weak assurance.
2) If the product has open source and design, we can redo the validation
3) If there is a published formal analysis, we can confirm it.

Alternatively, we could rely on a trusted certification process to certify the product. Two influential processes are the DOD TCSEC and the Common Criteria.

ORANGE BOOK CERTIFICATIONS

D – No security requirements

C1 – protected mode OS, authenticated login, DAC, security testing and documentation

C2 – DAC to level of individual user, object initialization, auditing

B1 – classification and Bell-LaPadula

B2 – system designed in top-down modular way, must be possible to verify, covert channels must be analyzed

B3 – ACLs with users and groups, formal TCB must be presented, adequate security auditing, secure crash recovery

A1 – Formal proof of protection system & model correctness, demonstration that implementation conforms to model, formal covert channel analysis

EXAMPLES

- NT4 was certified as C2 in a "special" configuration:
  - No network connectivity
  - No external media
  - Many extra security features turned on
- Most UNIX systems: C1 out of the box.
- B1 Unixes: HP-UX BLS, AT&T SysV/MLS, Trusted IRIX…
- B2 OSes: Multics (1985), Trusted XENIX (1993)
- B3: Wang XTS-300 (SysV; 2000)
- A1: SCOMP (1984)

COMMON CRITERIA

- CC has *protection profiles*:
  - Rationale – threat model, policies supported
  - Functionality Requirements – security features, services, and mechanisms required
  - Developer Assurance Requirements – Documentation, Guidance, etc.
- Each certificate has an Evaluation Assurance Level – review, testing, verification…
- The CC replace TCSEC, and are endorsed by 14 countries
- Evaluation is done by certified, independent companies, but paid for by the developer.

www.radium.ncsc.mil/tpep/library/ccitse/CCITSE-2.10-2.pdf

PROTECTION PROFILES

… are Common Criteria requirements for categories of systems, e.g. “Secure OS in a commercial environment.” They can themselves be certified under the CC.

An example protection profile is “Controlled Access” (CAPP_V1.d), intended to be the equivalent of TCSEC C2. CAPP_V1.d:

- **Assumes non-hostile, well-managed users**
- **Excludes hostile, well-funded attackers and malicious sys admins or developers**
- **Includes functional requirements like Authentication, User Data Protection, Prevent Audit Loss**
- **Includes Developer Assurance requirements like Security testing, Admin guidance, Life-cycle support**

Win2K was certified under CAPP_V1.d at EAL 4.