

COMP 435 Quiz 3 Study Guide

*Computer Security Concepts · UNC Fall 2026 · Operating systems,
access control and passwords*

0. What's on the quiz

WHEN. Friday, October 9, 2026, in class. 50 minutes.

FORMAT. Short answer on paper. Expect to apply a model to a figure, spot a vulnerable line of code, explain why a defense does or doesn't work, and do one or two back-of-envelope calculations.

SCOPE. L14 through L20 — the operating system, who is allowed to do what, and how we prove who someone is.

0.1 The four announced topics

Topic	What you need to be able to do	Section
OS security & access control	List the OS's responsibilities; distinguish RUID from EUID; explain privilege boundaries; say what a capability is and why possession conveys authority	§1, §3, §5
Security models	Tell a confidentiality policy from an integrity policy; read off who may read and who may write under Bell-LaPadula and Biba	§4
Race conditions	Trace a TOCTTOU window — check → state changes → use; explain why it becomes a vulnerability; name the fix	§2
Password security	Compute a search space; separate online from offline guessing; explain how hashing verifies a password; say exactly what a salt does and does not do	§6

Authentication concepts and biometrics were covered in L19–L20 but are not on the topic list. §7 covers them compactly — read it last.

0.2 Where each topic came from

Lecture	Title	Core ideas
---------	-------	------------

L14	Operating Systems Security	OS responsibilities; user vs. kernel mode; styles of separation; fence, base-bounds, segments
L16	Race Conditions, Access Control	TOCTTOU and the symlink attack; subjects, objects, rights; DAC mechanisms; MAC, Bell-LaPadula, Biba, reference monitors
L18	Password Guessing	online vs. offline; targeted, trawling, credential stuffing; hashing, precomputed dictionaries, iteration, salts
L19	Confused Deputy & Capabilities	RUID vs. EUID; setuid; ambient authority; capabilities; identification vs. authentication
L20	Authentication, Biometrics	password search space; the four means of authentication; biometrics, thresholds, tokens, 2FA

0.3 How to use this guide

Each section ends with a **Check yourself** set. §8 is a mock quiz — do it closed-book with a timer before reading §9, which works through the real practice set. §10 is a one-screen cram sheet.

EXAM TIP. Four answer shapes cover most of this quiz:

- **Apply a model to a picture.** Two people, one file, three levels. Decide read or write, then apply the arrow rule. Say which model you're using and in one clause why.
- **Find the vulnerable line.** Look for a place where *the caller supplies a name* and *the program supplies the privilege*, or where a check and a use are separated in time.
- **Explain what a defense does *not* do.** Salts, rate limits and process isolation each solve one specific problem and leave a neighbouring one untouched. Naming the leftover gap is usually the whole point.
- **Estimate a search space.** Count the alphabet, raise to the length, halve it for the average case, divide by the guess rate.

L14 · L19 1. The operating system and privilege

How does the operating system create isolation and manage access? Everything else on this quiz is a special case of that question.

1.1 The five responsibilities of the OS

Responsibility	Syscall examples	Breaks down into
Manage resources	<code>fork()</code> , <code>mmap()</code>	allocation · sharing · protection
Provide an interface to hardware	<code>read()</code> , <code>write()</code>	memory · file system · device I/O
Provide user authentication	<code>getuid()</code> , <code>setuid()</code>	user accounts · roles · passwords
Mediate interprocess communication	<code>pipe()</code>	shared memory · message passing
Protect itself	<code>kill()</code> , <code>mprotect()</code>	data · code · permissions

WHY THESE ARE SECURITY RESPONSIBILITIES. All five work in service of security policies. "Process A is authorized to access memory region A but not memory region B" is a policy, and resource management, hardware mediation and self-protection are what make it true.

The last one is not vanity. The data structures and code the OS uses to enforce access control, process separation and fair time sharing all live in memory, so they need protecting from everything else running on the machine.

1.2 User mode vs. kernel mode

System call.

The mechanism by which a program in user space asks the OS to do something on its behalf

The CPU runs in either **user mode** or **supervisor (kernel) mode**. Supervisor mode unlocks more instruction types and more memory ranges. Two rules:

- The processor should **never** execute user-provided code with supervisor privileges.
- **System calls are the mechanism for mode transition.**

WHY A SYSCALL IS SAFE AND A DIRECT JUMP WOULD NOT BE. A system call transfers control to a *designated kernel entry point*. The kernel then validates the request and its arguments, does the permitted work, and returns to user mode. The caller gets a **service**, not **privileges** — their own code never runs with kernel authority.

That distinction — service versus privilege — is the answer to any question of the form "why not just let the program do it itself?"

1.3 RUID vs. EUID

Real user ID (RUID).

The ID of the user who started the process

Effective user ID (EUID).

The user ID the OS uses to decide what permissions the program has while it runs

Most of the time EUID == RUID. The exception is the **setuid bit**:

SETUID. When the setuid bit is set on an executable and a user runs it, the **EUID becomes the file owner's UID** (usually root) while the **RUID stays the user's**. This lets certain programs perform privileged operations on behalf of ordinary users.

suid	guid	sticky		r	w	x		r	w	x		r	w	x
				Owner				Group				Users		

A legitimate use: `passwd` needs to write `/etc/shadow`, which you cannot write, so it runs `setuid-root`. The problem is that it now has *two* sets of permissions at once — yours and root's — and nothing in the mechanism tells it which one to use for a given operation. That gap is the confused deputy attack in §5.

EXAM TIP. If a question mentions "the executable has the setuid bit set," it is signalling a confused deputy problem. Expect to be asked which line is vulnerable and what mechanism fixes it.

1.4 Memory protection mechanisms

Mechanism	How it works	Limitation
Fence	A hardcoded address in the processor marking the start of OS code. Below is OS, above is user.	Protects the OS from users but not users from each other, and cannot adapt to a different memory layout.
Fence register	The fence becomes configurable, so the boundary can move.	Still one boundary — still only OS vs. user.
Base–bounds registers	A base and a bound delimit the current process's region, keeping one process out of another's memory.	One contiguous region per process; no per-region permissions.
Segment addressing	A table of segment descriptors per process: <code>base addr · length · R W X M F</code> , addressed as <code><segment #, offset></code> . Introduces virtual memory and per-region permissions.	More complex, and the descriptor tables are themselves protected state that must be defended.

THE SENTENCE TO WRITE. Base-bounds and segmentation are cases of **hardware and the operating system working together to protect one process's memory from another**. The OS populates the registers and tables; the hardware enforces them on every access. Neither could do it alone.

WATCH OUT. "A process enforces its own isolation" is always wrong. Self-enforced isolation is not enforcement — a malicious process simply declines. Isolation must come from a layer the process cannot reach, which is why these registers must be writable only by privileged code.

This is also why letting a program set its own base-bounds registers is catastrophic: it could widen its range to cover another process's memory or the kernel's, and then read secrets or corrupt the OS's own protection data. The registers are the enforcement; handing them to the enforced party ends the enforcement.

1.5 Four styles of separation

Style	Trade-off	Examples
Physical	not efficient, not complex, secure	air gapping; separate machines; a hardcoded memory fence
Temporal	more efficient, more complex, less secure	time sharing; object sanitization — reuse requires sanitizing
Logical (algorithmic)	more efficient, more complex, less secure	a configurable fence register; browser tab isolation; process address spaces
Cryptographic	issues of key management	encrypting each record under a different recipient's key

EXAM TIP. Ask **what is doing the separating?** Different hardware → physical. Different time slots → temporal. Software or addressing rules → logical. Different keys → cryptographic. Efficiency and complexity rise down the list; security falls.

1.6 Five features of a trusted system

TCB.

should be small and well defined

Trusted path.

authenticates the OS to the user

Secure start-up.

starts in a known good state

Object sanitization.

no object reuse — clear memory before handing it to someone else

Auditing.

an audit log tracks any changes

Two of these are counterintuitive enough to be worth a sentence each.

Trusted path runs the *opposite* direction from the usual authentication story. Passwords authenticate the user to the system; a trusted path authenticates the system to the user, so a fake login prompt drawn by a malicious program cannot harvest your credentials.

OBJECT SANITIZATION — THE ONE THAT GETS EXAMINED. Process isolation stops two *running* processes from reading each other's memory. It does nothing about memory that has been freed and handed to someone new. If the OS gives a dead process's pages to a fresh process without clearing them, the new process reads the old secrets from memory it is now *legitimately* allowed to access — no isolation rule was broken at all. The fix is object sanitization: zero the memory before exposing it. Merely freeing it is not enough.

1.7 Check yourself

1. Name the five OS responsibilities with a syscall for each. [L14]

Manage resources (`fork`, `mmap`); provide an interface to hardware (`read`, `write`); provide user authentication (`getuid`, `setuid`); mediate interprocess communication (`pipe`); protect itself (`kill`, `mprotect`).

2. A program runs `setuid-root`. Its **RUID** is 1001 (alice) and its **EUID** is 0. Which ID does the kernel use to decide whether it may open `/etc/shadow`? [L19]

The **EUID** — 0, root — so the open succeeds. The RUID still records that alice started it, which matters for accountability and for the program's own logic, but not for the permission check. That mismatch is exactly what makes `setuid` programs dangerous: the kernel answers "root wants this" when the truth is "alice asked for this."

3. Why is object sanitization a *temporal* separation problem rather than a logical one? [L14]

Because the resource is reused across time, so the only thing separating the old owner from the new one is *when* each held it. Logical separation (address spaces) works fine while both are running; it has nothing to say about a handover. Any separation that depends on sequencing rather than on a boundary needs explicit cleanup at the handover point.

4. A student proposes letting user programs write their own base-bounds registers so they can request memory without a syscall. What goes wrong? [L14]

A malicious process widens its range to include another process's memory or the kernel's, then steals secrets or corrupts OS code and protection data. The registers *are* the isolation mechanism, so a process that can set them has no isolation. They must be controlled by privileged OS code with hardware enforcing the limits, and a system call is how a program requests more memory without gaining kernel privileges — the kernel validates the request, does the work, and returns to user mode.

L14 · L16 2. Race conditions and TOCTTOU

A race condition is a vulnerability class, and the announced topic asks for three things: trace the window, explain why it becomes a security problem, and name the fix.

2.1 What a race condition is

RACE CONDITION. Concurrent access to a resource is not serializable. It occurs when two or more processes have access to the same resource, and the final state of the resource depends on the relative timing of the two.

Balance starts at 900; two withdrawals interleave:

ATM 1	Bank	ATM 2
	balance = 900	
w/d \$100		w/d \$50
	get_bal() → 900	get_bal() → 900
\$100 out		\$50 out
	set_bal(800)	set_bal(850)
	balance = ??	

The customer withdrew \$150 and the balance ends at 800 or 850 depending on which write lands last. Either way the bank loses money.

WHERE THE BUG ACTUALLY IS. Not "two users accessed the account at once" — that is normal and must be supported. The bug is that the supposedly single operation **withdraw** was implemented as **multiple interleavable operations**. The fix is locking or transactions, so only one withdrawal updates the balance at a time.

WATCH OUT. If you are asked to identify the fix, say **make the operation atomic** — or equivalently, avoid a separate check-and-use. Answers like "add more permission checks" or "run the processes in a fixed order" miss it: the problem is the gap, not the permissions and not the scheduling.

2.2 TOCTTOU

TIME OF CHECK TO TIME OF USE (TOCTTOU). A particular type of race condition in which the gap between *checking* permissions and *using* permissions can be exploited.

check	state changes	use
(access is OK)	(attacker swaps the thing)	(the program acts on a decision that is no longer true)

The decision was correct when it was made. By the time it is relied on, the world has changed underneath it.

2.3 The symlink attack

A symbolic link redirects a path at open time: after `symlink(target, linkpath)`, opening `linkpath` follows the link and operates on `target`.

Now take a privileged program that writes to a temporary file:

Program A — privileged

```
filename = "tmpname";

// write to the file
fd = open(filename,
           O_CREATE|O_RDWR);
```

Program B — attacker

```
symlink("/etc/passwd", "tmpname");
```

If the attacker wins the race and plants the symlink between the moment the privileged program decides on `tmpname` and the moment it opens it, the privileged program opens `/etc/passwd` instead, with its own privileges.

2.4 The `access()` version

The more explicit form, and the one most likely to appear on paper:

```
if (access(fname, W_OK) == 0) {      // CHECK: does the *real* user have write permission?

    fd = open(fname, O_WRONLY);      // USE: open it with the *effective* privileges
}
```

`access()` returns 0 when the *current real user* has write permission. A `setuid` program calls it precisely to avoid the confused deputy problem — "don't use my root powers on a file the caller couldn't open themselves." The intention is correct. The implementation leaves a window:

```
if (access(fname, W_OK) == 0) {
    // ATTACKER, in the gap:
    //  unlink(fname);
    //  symlink("/etc/passwd", fname);
    fd = open(fname, O_WRONLY);    // now opens /etc/passwd as root
}
```

THE INVARIANT BEHIND EVERY RACE CONDITION. Two entities have access to the same resource, and one of them can change it between another's check and use.

EXAM TIP. The exploitable gap is between **two separate syscalls that both resolve the same pathname**. The fix is not a better check — it is to stop checking and using separately: open the file once and operate on the returned file descriptor, which is bound to the actual object rather than to a name that can be re-pointed. (Dropping privileges around the operation, or using an `openat`-style call that takes a directory handle, are the same idea.)

2.5 Which principle does TOCTTOU violate?

COMPLETE MEDIATION. Every access to a protected object should be checked. A TOCTTOU bug checks *once* and then relies on that decision at a later moment when it may no longer hold — so the access that actually happens was never mediated.

This connects back to the access control best practices in §3.2: "universal application" is the same requirement, and caching an authorization result is exactly the thing that makes it fail.

2.6 Check yourself

1. Mark the check, the window, and the use:

```
if (stat(path, &st) == 0 && st.st_uid == getuid()) {
    unlink(path);
}
```

[L16]

Check: `stat` confirms the file at `path` is owned by the caller. **Window:** between `stat` returning and `unlink` running, the attacker replaces `path` with a link to a file they do not own. **Use:** `unlink` resolves `path` a second time and deletes the wrong file. Both calls resolve the name independently, which is the defect.

2. Why doesn't adding a second `access()` check immediately before `open()` fix the bug?

[L16]

It narrows the window without closing it. There is still a moment between the final check and the open in which the attacker can swap the target, and an attacker who can lose the race can simply retry — these attacks are run in a loop. Any fix that leaves a gap at all is not a fix; you have to make check and use a single indivisible operation on a single object.

3. A race condition causes a bank balance to be wrong. A different race condition lets an unprivileged user overwrite `/etc/passwd`. Why is only one of them a *security* vulnerability? [L16]

Both are correctness bugs; the second is a security bug because the timing gap lets an attacker cross a **privilege boundary** — an action that should have required root happens at the attacker's direction. A race becomes a security vulnerability when the state that changes during the window is the state an authorization decision depended on. (The bank case is still a security problem if an attacker can trigger the interleaving deliberately, which is the point of the lab.)

4. Name the fix in one phrase, and say what it is *not*. [L16]

Make the check and the use atomic — one indivisible operation, via locking, a transaction, or operating on a handle obtained once rather than re-resolving a name. It is *not* more checks, not a stricter permission model, and not ordering the processes: the resource is shared and the gap is the vulnerability.

L16 3. Access control

3.1 Subjects, objects and rights

ACCESS CONTROL. Which **subjects** can access which **objects**, and with which **rights**.

Three slots, and every access control question is about filling them in:

Slot	Question	Examples
Subject	Who?	users, processes
Object	What?	files, memory, I/O devices, users, processes
Right	How?	read, write, execute, create, transfer

Users and processes appear in both lists. A process is a subject when it opens a file and an object when another process tries to kill it, and which role it is playing depends entirely on the direction of the access being checked.

3.2 Three best practices

Universal application.

Every access of an object by a subject should be checked

Least privilege.

Every subject should be granted the least amount of access necessary to do its job

Type checking.

Operations should be meaningful for the object accessed

Universal application is complete mediation under another name, and it is honoured mostly in the breach — using cached authorization results is a bad idea, and we use cached results all the time. The weakened versions:

- **random checking** — check a randomly selected subset of accesses
- **random auditing** — log every access, then randomly check some later. This provides *detection*, not *prevention*.
- **selective checking** — if a cheap first glance looks suspicious, do a thorough check

Least privilege supports failing safe: limiting a subject's privilege limits the damage if it is compromised or turns out to be malicious. Its practical difficulty is that you need to know up front exactly what access each subject will need, so designers routinely *over*-privilege a process to guarantee it never breaks — which is how you end up with setuid-root binaries that need one specific capability.

Type checking is a type system in the security sense: it defines which kinds of objects exist and which operations are legal on them, so "execute this data file" or "seek on this network socket" is rejected as meaningless rather than merely unauthorized.

3.3 Two mechanisms for authorization decisions

When something asks "is this process authorized to access this resource?", there are two families of answer: **access control** (consult a table indexed by subject and object) and **capabilities** (the subject presents an unforgeable token). Capabilities get their own treatment in §5, because the reason they exist is the confused deputy attack.

3.4 Four policy families

Policy	Who decides permissions
Discretionary (DAC)	The owner of the object, at their discretion
Role-Based (RBAC)	Permissions attach to roles; users are assigned roles
Attribute-Based (ABAC)	Permissions follow from attributes of subject, object and context
Mandatory (MAC)	A system-wide policy the owner cannot override — see §4

The word that separates DAC from MAC is **discretion**. Under DAC, if Alice owns a file she may grant anyone access to it. Under MAC she may not, because the policy is not hers to change.

3.5 Three ways to store a DAC policy

All three encode the same matrix; they differ in how it is sliced and therefore in what is cheap.

Mechanism	What it is	Pros	Cons
Access control matrix	The full subjects $\times$ objects table	easy · revocation is easy · no aliasing	sparse matrix — mostly empty cells, so it wastes space
Access control directory	One list per <i>subject</i> : what this user may do, object by object	easy · fixes the sparsity · default rights	file revocation is hard (you must visit every user's list) · aliasing
Access control list (ACL)	One list per <i>object</i> : who may do what to this file	file revocation is easy · no aliasing · default rights	user revocation is hard (you must visit every file)

EXAM TIP. The directory/ACL trade-off follows from which way you sliced the matrix. Slice by subject and you can answer "what can Bob do?" instantly but must search everywhere to revoke access to one file. Slice by object and it reverses. Nothing is free; you are choosing which question is cheap.

3.6 What a DAC matrix cannot do

Take this policy for a shared instructional machine:

	exam	syllabus	printer
Alice (instructor)	read, write	read, write	print
Bob (TA)	read	read, write	print
Carol (student)	—	read	print

The matrix enforces the stated policy exactly. And yet:

THE VIOLATION. Bob may read the exam. Bob may write the syllabus. So Bob copies the exam's contents into the syllabus — both operations permitted. Carol may read the syllabus, so Carol now has the exam. **Every individual access was authorized by the matrix, and the exam's confidentiality is gone.**

The matrix controls *direct* accesses. It says nothing about where information goes once a subject legitimately holds it, so it cannot prevent this indirect flow. That gap is the entire motivation for mandatory access control: MAC's goal is to understand and control how information *flows* through a system, not merely who may touch what.

WATCH OUT. A question asking you to "violate the spirit of the policy despite correct enforcement" wants this answer, and it wants a specific two-hop path: a subject who can read something sensitive and write somewhere less sensitive, plus a third party who can read the destination. Name the people and the two operations; don't just say "information could leak."

Note that Bob does not have to be malicious. The same leak happens by accident if he pastes a question into the syllabus as an example, which is why the structural answer — the policy permits it — matters more than anyone's intent.

3.7 Check yourself

1. In "process A kills process B," name the subject, object and right. [L16]

Subject: process A. Object: process B. Right: the kill/signal right. Processes are both subjects and objects; which one B is depends on the direction of this particular access.

2. A system logs every file access and an admin reviews a random sample weekly. Which best practice is being approximated, and what has been given up? [L16]

Universal application, approximated by **random auditing**. What is given up is prevention — the policy is not enforced at access time, so a violation is at best detected after the fact. Detection is worth having, but it is not mediation.

3. An admin must remove a departing employee's access to everything. Is this easier with an access control directory or with ACLs? [L16]

Easier with a **directory**: that user's list is one object, so deleting it revokes everything at once. With ACLs you must visit every file to find and remove entries naming them — user revocation is the ACL's weak case, just as file revocation is the directory's.

4. Would least privilege alone have prevented the Bob-copies-the-exam leak? [L16]

No. Bob genuinely needs read on the exam and write on the syllabus to do his job, so neither right is excess privilege. The leak comes from *combining* two necessary rights, which no per-right minimisation can catch. You need a policy that constrains information flow between levels — mandatory access control — rather than one that constrains individual accesses.

L16 4. Security models: Bell-LaPadula and Biba

Mandatory access control exists because of the gap in §3.6. Its goal is to **understand and control how information flows through a system**, not merely who may touch what.

Two key features: it is a **multi-level security (MLS)** system, and it is enforced by a **reference monitor**.

4.1 Levels, clearances and classifications

high	top secret
	secret
	confidential
	restricted
low	unclassified

Clearance.

The level assigned to a **subject** — a person or process

Classification.

The level assigned to an **object** — a file

Mandatory means mandatory: a subject cannot hand out access to an object the way they could under DAC, because the levels, not the owner, decide.

4.2 The two models

	Bell-LaPadula	Biba
Protects	Confidentiality	Integrity
Reading	no read up	no read down
Writing	no write down	no write up
So information flows...	upward only — secrets can never descend	downward only — contamination can never ascend

BELL-LAPADULA, IN WORDS. You may not *see* anything above you, and you may not *spill* anything below you. A Secret user cannot read a Top Secret file (no read up) and cannot write into a Confidential file (no write down), because writing down is how a secret escapes to someone not cleared for it.

BIBA, IN WORDS. You may not *contaminate* anything above you, and you may not *drink* from below. A Secret-integrity process cannot write into a Top Secret file (no write up) and cannot read an Unclassified file (no read down), because reading down is how bad data gets into a trusted computation.

The two are mirror images, which is why they are easy to confuse and why they are examined together. If you can only remember one thing, remember **Bell-LaPadula = no read up, no write down**, and get Biba by flipping both.

EXAM TIP. Work it in three steps every time. **(1)** Which model — confidentiality or integrity? **(2)** Is the operation a read or a write? **(3)** Is the subject above or below the object? Then apply the rule. Writing those three facts down before answering costs ten seconds and prevents the usual mirror-image mistake.

4.3 Worked example

Purple has **Top Secret** clearance. Orange has **Unclassified** clearance. The file is classified **Secret** — so Purple is above it and Orange is below it.

Question	Purple (above)	Orange (below)	Answer
Under Biba , who may read the file?	reading down — forbidden	reading up — allowed	Orange only
Under Bell-LaPadula , who may write the file?	writing down — forbidden	writing up — allowed	Orange only

Both answers come out "Orange only," which feels wrong the first time and is a useful check that you applied the rules rather than your intuition. Orange is the low subject, and the two *permitted* operations for a low subject are reading up (Biba) and writing up (Bell-LaPadula). Purple is the high subject, and both of the operations asked about are downward, which is precisely what each model forbids.

WATCH OUT. Intuition says the Top Secret person should be allowed to do more. Under Biba they are allowed to do *less* with lower-level data, because Biba is not about secrecy at all — a high-integrity process reading a low-integrity file is the risk it is designed to stop. Levels in a Biba question are integrity levels even when the labels say "Top Secret."

4.4 Reading at your own level only

What if a model let subjects read and write only at their own level? That satisfies no-read-up and no-write-down (so confidentiality holds) *and* no-write-up and no-read-down (so integrity holds). The answer is **confidentiality and integrity** — but not availability, which neither model addresses. Enforcing both at once is possible; it is just extremely restrictive, since nothing can ever move between levels.

4.5 The reference monitor

REFERENCE MONITOR. Monitors execution and enforces the access control policy. It must be **complete mediation** (every single access passes through it), **tamperproof**, and **verifiable**.

Those three properties are why the reference monitor is part of the TCB: security relies on it, and it is permitted to see every access in the system. Modern operating systems don't implement one in the strict sense, but a **network firewall** is a real-world example — all traffic from outside must pass through it.

WHICH PRINCIPLES DOES MAC REST ON?. **Principle of least privilege and complete mediation.** Subjects receive only the access needed for their work, and every access is checked against the policy. The reference monitor supplies the complete mediation; the level assignments supply the least privilege.

4.6 The analog hole

Purple holds a Secret file that Orange may not read. Orange asks what it says. Purple says, "It says Orange is a spy."

No rule was broken inside the system, and the information arrived anyway. A security model governs the channels it mediates; it cannot govern a human who reads the screen and talks, a phone camera, or a printout. This is worth a sentence in any answer about the limits of a model — enforcement ends where the monitored system ends.

4.7 Check yourself

1. A Confidential-clearance analyst wants to read a Top Secret report. Allowed under Bell-LaPadula? [L16]

No — that is reading up, which Bell-LaPadula forbids. The subject is below the object and the operation is a read, so the confidentiality rule blocks it.

2. Same analyst wants to write into a Top Secret report. Allowed under Bell-LaPadula? Under Biba? [L16]

Bell-LaPadula: yes — writing up is permitted, since pushing information to a more-classified place does not disclose it. **Biba: no** — writing up is exactly what Biba forbids, because low-integrity data would contaminate a high-integrity object. The same operation is required by one model and prohibited by the other, which is why systems rarely enforce both fully.

3. Why can't Bell-LaPadula stop the Bob-copies-the-exam leak from §3.6 by accident?

[L16]

It can — that is the point of it. Label the exam Secret and the syllabus Confidential, and Bob, cleared at Secret, may read the exam but may not write down into the syllabus. The flow the DAC matrix permitted is now structurally impossible, because the rule constrains where information may go rather than which individual operations are allowed.

4. A reference monitor is tamperproof and verifiable but checks only every tenth access.

Which property fails and what is the consequence? [L16]

Complete mediation fails. Nine in ten accesses are unmediated, so the policy is no longer enforced — it is sampled. This is the "random checking" weakening from §3.2, and it reduces the monitor from a prevention mechanism to a detection one.

L19 5. The confused deputy attack and capabilities

This is the single most likely place to lose marks, because the question format is "find the vulnerable line, explain it, name the fix" and all three parts have precise answers.

5.1 Ambient authority

AMBIENT AUTHORITY. All the extant authority of the current execution context.

The problem is that authority **accrues** as a program runs. A setuid program holds its own privileges and acts on requests from a less-privileged caller, so at any moment it has multiple sets of permissions available — and when it opens a file by name, nothing in the operation says *on whose behalf* it is acting. It just uses whatever authority it has.

5.2 The compiler example

A compiler that runs setuid, so it can append usage statistics to a protected system file:

```
int main(int argc, char *argv[]) {
    /* compile user's source code */

    // write out binary executable
    FILE *fp = fopen(argv[2], "w");           // ← the caller names this file
    /* write compiled program to fp */

    // append compiler usage stats to a protected system file
    fp = fopen("/etc/compiler_stats", "a");    // ← the compiler needs privilege for this
    /* write usage stats to fp */
}
```

Used normally:

```
$ gcc prog.c -o prog
```

Used maliciously:

```
$ gcc prog.c -o /etc/passwd
```

The compiler opens `/etc/passwd` for writing **using its own root privileges** and truncates the password file — even though the user who ran it had no permission to touch it.

CONFUSED DEPUTY ATTACK. A program running with multiple sets of permissions uses its ambient authority indiscriminately.

Role	Who
Deputy	The compiler — it has authority to write system files
Attacker	The unprivileged user, who supplies <code>/etc/passwd</code> as the output filename
Victim	The system, or the admin who trusts the compiler
The confusion	The compiler doesn't realise it is acting <i>on behalf of a user</i> when it opens that file. It just uses its own privilege.

EXAM TIP. The vulnerable line is always the one where **the caller controls the pathname and the program supplies the privilege**. Scan the listing for a `fopen / open` whose argument came from `argv`. The line that writes to a hardcoded system path is *not* the bug — that one is doing what it was designed to do.

Writing the explanation, there are three clauses worth including: the caller controls the pathname; the program opens it with *its own* privileges; so an unprivileged caller can name a protected file and have it truncated and written to. Then name the role: the program is the deputy, incorrectly using its own authority for a caller-directed operation.

5.3 Why access control can't fix this

THE KEY LIMITATION. Access control cannot prevent the confused deputy attack.

Think about what the kernel sees. A process with EUID 0 asks to open `/etc/passwd` for writing. Root may write that file. The access control check passes, correctly — the matrix was consulted and the answer really is yes. Nothing in the request records that the pathname came from an unprivileged user, because **a pathname carries no authority with it**. The subject is the process, and the process is privileged. The check cannot distinguish "root wants this" from "alice asked root for this."

5.4 Capabilities

CAPABILITY. An object reference *plus* an access right — a "fat pointer." It says both **where an object is** and **what you are allowed to do with it** (read, write, execute, grant).

Properties:

- **unforgeable token** — you cannot manufacture one for an object you were not given
- **the token directly ties the access right to the object**
- **no designation without authority** — you cannot even *name* a thing you have no right to

And therefore:

- **no ambient authority** — every operation says which authority it is using

- supports the principle of least privilege

The compiler rewritten with capabilities:

```
int main(int argc, char *argv[]) {
    /* compile user's source code */

    // write out binary executable - uses the USER's capabilities
    cap *fcap = get_cap(argv[2], user_dir_cap);
    FILE *fp = cap_fopen(fcap);
    /* write compiled program to fp */

    // append usage stats - uses the SYSTEM's capabilities
    fcap = get_cap("/etc/compiler_stats", sys_dir_cap);
    fp = cap_fopen(fcap);
    /* write usage stats to fp */
}
```

```
$ gcc prog.c -o /etc/passwd
> ERROR: no capability for passwd file!
```

The attack now fails at `get_cap`, because the lookup is performed against `user_dir_cap` — the caller's authority — and the caller has no capability for `/etc/passwd`. The privileged operation uses `sys_dir_cap` explicitly and separately.

WHY THIS WORKS, IN ONE SENTENCE. Possession conveys authority, so the program has to say *whose* authority each operation uses — and the caller's pathname alone can no longer unlock the deputy's privileges.

EXAM TIP. "What mechanism is designed to stop confused deputy attacks?" → **Capabilities.** If asked to elaborate: require a capability granting write access to the caller's chosen output file, and use a *separate* capability for appending to the protected file. A pathname alone must not let the caller exercise the program's privileged authority.

5.5 Check yourself

1. In this `setuid-root` listing, which line is vulnerable and why?

```
1 FILE *fp1, *fp2;
2 // open argv[2] for writing
3 fp1 = fopen(argv[2], "w");
4
5 // open fstab file to append
6 fp2 = fopen("/etc/fstab", "a");
```

[\[L19\]](#)

Line 3. The caller controls the pathname via `argv[2]`, but the program opens it with its root privileges. An unprivileged caller can supply a protected file's path, causing the program to truncate it and write error messages into it. The program is the deputy, incorrectly using its own authority for a caller-directed operation. Line 6 is a hardcoded privileged path and is not the flaw.

2. Why doesn't running the program as a less-privileged user fix it? [\[L19\]](#)

Because the program genuinely needs the privilege for its other job — appending to the protected system file. Drop the privilege and that breaks. The problem is not that the deputy *has* authority but that it cannot tell which of its two authorities a given operation should use, so removing the authority removes the function rather than the confusion.

3. A capability is described as unforgeable and as providing "no designation without authority." What would go wrong if capabilities were forgeable? [\[L19\]](#)

They would collapse into pathnames. The guarantee is that holding a capability *is* the authority, so if an attacker can mint one for any object, every object becomes reachable by anyone and the mechanism provides nothing. "No designation without authority" is what closes the confused deputy hole: you cannot hand the deputy a reference to a file you had no right to name.

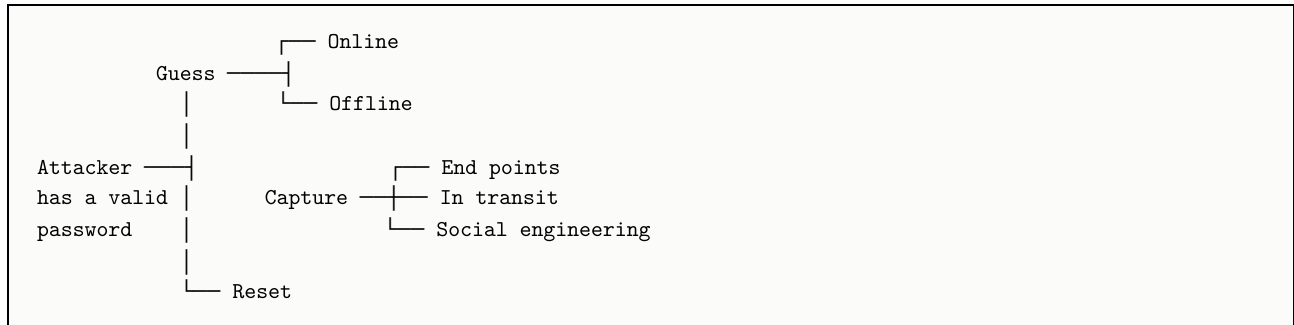
4. Both the confused deputy attack and TOCTTOU involve a privileged program misusing its authority on a caller-supplied pathname. What is the difference? [\[L16 · L19\]](#)

In the **confused deputy**, nothing changes during execution — the program never checks whose behalf it is acting on, so the attack works on the first try with no timing involved. In **TOCTTOU**, the program *does* check correctly, but the attacker alters the state between the check and the use. One is a missing check; the other is a check that was true and then stopped being true. The fixes differ accordingly: capabilities for the first, atomicity for the second.

L18 · L20 6. Password security

Four things to be able to do: compute a search space, separate online from offline guessing, explain how a hash verifies a password, and say precisely what a salt does and does not do.

6.1 How passwords get attacked



Everything below is the **Guess** branch. Capture and reset matter too, but the quiz topic is guessing.

6.2 Password search space

An 8-character password drawn from the printable characters excluding space: 52 alphabetic + 10 numeric + 32 additional = **94** characters.

$$94^8 = 6,095,689,385,410,816 \approx 6.1 \times 10^{15} \text{ possible passwords}$$

$$\text{chance of guessing right on one try} = 1 / 94^8 \approx 1.6 \times 10^{-16}$$

$$\log_2(94^8) = 8 \times \log_2(94) \approx 8 \times 6.55 \approx 52.4 \rightarrow \text{about } 2^{53}$$

Now: if there are 2^{53} possible passwords and the attacker can try 10,000 per second, how long on average?

$$\begin{aligned} \text{average guesses needed} &= 2^{53} / 2 \approx 4.50 \times 10^{15} \\ \text{time} &= 4.50 \times 10^{15} / 10^4 \approx 4.50 \times 10^{11} \text{ seconds} \\ &\approx 14,300 \text{ years} \end{aligned}$$

EXAM TIP. Three steps, and the middle one is the one people drop: **(1)** alphabet size to the power of the length, **(2)** *halve it*, because on average you find it halfway through, **(3)** divide by the guess rate. Converting to a power of 2 is just $\text{length} \times \log_2(\text{alphabet})$, and $\log_2(94) \approx 6.55$.

The same arithmetic covers a key search. For a 56-bit key there are $2^{56} \approx 7.2 \times 10^{16}$ possibilities:

Machine speed	Full search	Average (half)
---------------	-------------	----------------

10^{10} encryptions/sec	$\approx 7.2 \times 10^6 \text{ s} \approx \mathbf{83 \text{ days}}$	$\approx 42 \text{ days}$
10^{15} encryptions/sec	$\approx \mathbf{72 \text{ seconds}}$	$\approx 36 \text{ seconds}$

WATCH OUT. These numbers say a *random* 8-character password is fine. The reason passwords fall anyway is that **people do not choose uniformly at random**, so the attacker never searches the full space — see §6.5. Any answer that stops at "14,000 years, therefore secure" has missed the point of the lecture.

6.3 Online guessing

The attacker submits guesses to the live login service and sees accept or deny. Three strategies:

Attack	Definition	How it works
Targeted	An attack aimed at a <i>specific user</i>	Guess common passwords, then use personal information about that person — pet names, family names, dates
Trawling	An attack that tries to gain access through <i>any</i> of a system's users	Pick one common password and try it against thousands of accounts until something hits
Credential stuffing	Reusing credentials breached from one site against another	Take working username–password pairs from a breach and submit them elsewhere

The countermeasure for targeted attacks is to **rate-limit guesses**. But note what a *per-account* rate limit does and doesn't do:

WHY RATE LIMITS BARELY SLOW A TRAWLING ATTACKER. A per-account limit restricts guesses against each account separately. The trawling attacker makes only a few guesses per account — staying comfortably inside every individual limit — while making an enormous number of guesses *in total* across thousands of accounts. They don't need to break into *your* account; they need to break into *an* account.

WHY CREDENTIAL STUFFING BEATS A STRONG PASSWORD. The attacker already has the password, so they submit it directly — there is no guessing to rate-limit. Length and randomness protect against *guessing*, and this is not guessing. Only using a different password on each site helps.

6.4 Offline guessing and hashing

The attacker steals the password database and now works on their own hardware.

HASHED PASSWORD. A hashed value is a fixed-length string produced by applying a hash function to input data. Easy to compute, hard to reverse. The server stores $h(\text{password})$ rather than the password, and verifies a login by hashing what you typed and comparing.

That verification procedure is also the attacker's procedure, which is the crux of the topic:

HOW AN OFFLINE GUESS IS CHECKED WITHOUT REVERSING THE HASH. The attacker applies the *same* password-hashing procedure to each candidate — using the stored salt and parameters if there are any — and compares the result against the stolen hash. A match means the guess is right. The hash function is never inverted; it is only ever run forwards, exactly as the login service runs it.

WHY THE RATE LIMIT EVAPORATES. Online guesses go through the login service, which can count and restrict them. Offline guesses run on the attacker's own hardware, so the service cannot see them, count them, or limit them. The only thing standing between the attacker and the password is the *cost of computing one hash*.

The weakness of plain hashing: hash algorithms are deterministic, so the same input always gives the same output. Two users with the same password get the same stored hash — visible by inspection in the stolen file — and worse, the attacker can precompute.

PRECOMPUTED DICTIONARY ATTACK. Hash a list of likely passwords once, store (password, hash) pairs, then look up every stolen hash in that table. The expensive work is done a single time and reused against every database and every account forever.

6.5 Two defenses

Iterated hashing

Store $h(h(h(\dots(\text{password}))))$ for d iterations instead of a single hash.

Effect on the attacker	Cost to the service
Each guess costs d times as much, so for fixed computing resources the guessing rate drops by a factor of d .	The service does the same extra work on every legitimate login , increasing login latency and server load.

That symmetry is the examinable part: iteration does not make the attacker's job qualitatively different, it makes it proportionally more expensive — and you pay the same multiplier. The parameter is chosen so that the cost is negligible once per login and painful a trillion times.

Salting

SALT. A string unique to that user, concatenated to the password before hashing. The salt is **not a secret** — it is stored in the database right next to the hash. It exists to make precomputation useless, not to hide anything.

Identity	Salt	$h(\text{password}, \text{salt})$
Jane	0xdef	0x93762d21
Patrick	0x32c	0x2874efa2
Philip	0xfef	0x954eabbc
Roz	0xcaf	0x5609ab1c

Two users who choose the *same* password now get *different* stored hashes, because the hash inputs differ. (A collision is possible but vanishingly unlikely with a decent hash function.)

WHAT A SALT DOES. It destroys the reuse of precomputed work. A dictionary built for one salt is useless for another, so the attacker must recompute candidate hashes **separately for every distinct salt** — turning one precomputation that cracks every account into one precomputation per account. The salts need not be secret for this to hold, because the cost is in the computation, not in the knowledge.

WHAT A SALT DOES NOT DO. It does not make a weak password safe. The attacker knows your salt — it's in the stolen file — so they hash common password guesses *with your salt* and compare against your hash. If your password is `qwerty`, it falls in milliseconds. Salting limits precomputation and the sharing of work across accounts; it does not make a guessable password unpredictable.

EXAM TIP. "A developer claims salting makes a common password safe" is a trap with a scripted answer: **no** — state that the attacker has the salt, can hash guesses with it, and compares; then state what salting *does* buy, which is limiting precomputation and cross-account reuse of work.

6.6 Why the math in §6.2 doesn't save you

Passwords are not chosen uniformly at random. Attackers start with the most common ones, and the real search space is tiny compared with 2^{53} .

Category	Size
qwerty, 12345678, password, 11111111	learnable online, a very small set

family names, pet names	small, and often publicly discoverable — the basis of a targeted attack
words in the dictionary	about 600,000 English words
common number substitutions	e→3, o→0, t→7, l→1, f→4 — multiplies the dictionary by a small constant, nothing like 94 ⁸

The requirement for a good password is genuinely in tension with itself: **chosen uniformly at random** and **easy to remember**. Essentially every password weakness follows from people resolving that tension in favour of memorability.

6.7 Check yourself

1. A service allows five failed logins per account per minute. An attacker steals the password database. Why does the limit stop one kind of guessing and not the other?

[L18]

Online guesses pass through the login service, which can count and refuse them. Offline guesses run entirely on the attacker's own hardware against the stolen hashes, so the service never sees them and has nothing to limit. Once the database is out, the rate limit is irrelevant and only hashing cost slows the attacker.

2. Two users pick the same password and the service salts each with a different salt. Should the stored hashes match? [L18]

No — the salts make the hash *inputs* different, so the outputs differ (an accidental collision is possible but very unlikely). This is what prevents an attacker from spotting shared passwords in the stolen file, and more importantly it means a table computed for one salt cannot be reused for the other.

3. An attacker tries Password1 against 50,000 accounts. Is this targeted or trawling, and why does a per-account rate limit barely help? [L18]

Trawling — the goal is access through *any* user, not a particular one. One guess per account stays far inside a five-per-minute per-account limit while producing 50,000 guesses system-wide. The limit constrains guesses per account; the attacker's budget is measured across accounts.

4. A user picks a 20-character random password and reuses it on two sites. One is breached. Is the other safe? [L18]

No — this is **credential stuffing**. The attacker has the working password and submits it directly, so there is no guessing for the length or randomness to defeat. Once exposed, a password's strength is irrelevant; only using a different password per site prevents the reuse.

5. A service raises its hash iteration count from 1,000 to 100,000. What changes for the attacker, and what does it cost the service? [\[L18\]](#)

Every guess costs $100\times$ more, so the attacker's guessing rate drops by $100\times$ for the same hardware. The service pays the same $100\times$ on every legitimate login — more latency per login and more CPU per authenticating user. The defense works precisely because the service hashes once per login and the attacker hashes billions of times.

L19 · L20 7. Authentication and biometrics (background)

Covered in L19–L20 but not on the announced topic list. Read it last, and read it for the definitions rather than the detail.

7.1 Identification vs. authentication

Identification.

Asserting who a person is — "Hello, my name is Kaki"

Authentication.

Proving an asserted identity — showing a driver's licence

Identification	Authentication
public	private
well known	unforgeable · authentic

A username is an identifier and is meant to be public; a password is an authenticator and must not be. A name tag identifies you and authenticates nothing, which is why it is the odd one out in a list that otherwise contains a licence, a passport and a door key.

7.2 The four means of authentication

Means	Example
What you know	passwords
What you do or what you are	biometrics
What you have	tokens, keys, credit cards
Where you are	location

TWO-FACTOR AUTHENTICATION. Requiring two *independent* means. A driver's licence is a nice illustration: the card itself is a physical token (what you have), and the photo is biometric (what you are).

Passwords are ubiquitous, easy to use and need no extra hardware — and they are easy to attack and difficult to use well. Every authentication system is a trade among those.

7.3 Biometrics

BIOMETRICS. Measurement of a physical characteristic — fingerprints, hand geometry, retina, iris, voice, facial features.

Authentication runs in two phases, **registration** (enrol a reference template) and **authentication** (capture a fresh sample and compare). Each phase has a characteristic failure:

Failure to enrol (FTE).

At registration, the sensor cannot capture an adequate sample

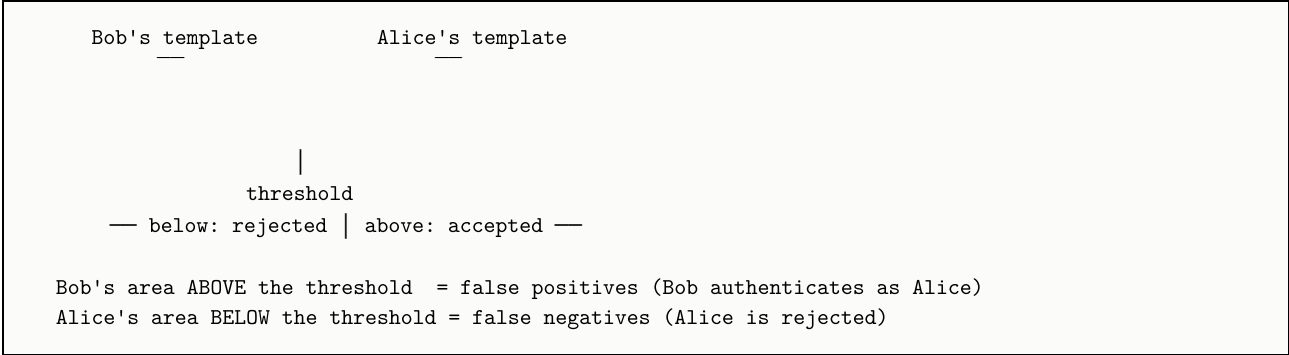
Failure to capture (FTC).

At authentication, the sensor cannot capture an adequate sample

REFERENCE TEMPLATE. A feature-based digital representation of a person's biometric trait. Comparison is never exact — it produces a *score*, and the score is compared against a threshold.

	Is the identified user	Is not the identified user
Considered a match	True positive	False positive
Considered not a match	False negative	True negative

Plot the score distributions for two people and they overlap. The threshold sits somewhere in that overlap, and moving it trades one error for the other:



EXAM TIP. "The probability of Bob falsely authenticating as Alice is the area of which region?" — the part of **Bob's** distribution that lies on the *accept* side of the threshold. Lower the threshold and false positives rise while false negatives fall; raise it and the reverse. There is no setting that eliminates both, because the distributions overlap.

	Pros	Cons
Usability	low cognitive load · nothing extra to carry · scales well for users	failure to enrol · failure to capture · false negatives · privacy concerns

Security	physical characteristics are unique	insecure in remote or unsupervised settings · cannot be revoked · false positives
-----------------	-------------------------------------	--

THE TWO THAT MATTER. Biometrics are **insecure when unsupervised or remote** — nobody is checking that the finger is attached to a living person, and across a network you receive only a claimed measurement. And they **cannot be revoked**: a leaked password is replaced in seconds, a leaked fingerprint is yours for life. Overall, security is lower than people expect.

7.4 Tokens, revocation and reset

AUTHENTICATION BY WHAT YOU HAVE. Possession of an item is sufficient for authentication — keys, credit cards, hardware tokens.

Proving possession *remotely* is the hard part, since you cannot hold the thing up to the screen. Two approaches: **challenge-response** (the server sends a challenge only the token can answer, which needs cryptography) and **synchronized passcodes** (both sides generate the same passcode independently and compare).

Revocation.

Cancelling a means of authentication — possible for passwords and tokens, not for biometrics

Resetting a password.

Authenticate as the user, then set a new password

That two-step reset is worth a second look: the reset path is itself an authentication, so it is only as strong as whatever it falls back on. A system with an excellent password policy and a security-question reset flow is as strong as the security questions.

7.5 Check yourself

1. Which is not a means of authentication: a driver's licence, a passport, a door key, or a name tag? [L19]

The **name tag**. It identifies — asserts who you are — but proves nothing, because anyone can write any name on one. The other three are things you have, and are at least nominally unforgeable.

2. Why does lowering the biometric threshold increase false positives? [L20]

The threshold is the score above which a sample is accepted. Lowering it moves the accept boundary left, so more of the *impostor's* score distribution falls on the accept side — more impostors are let in. The same move reduces false negatives, since less of the legitimate user's distribution falls on the reject side. The overlap means you are always choosing between the two.

3. A password and a security question are both required. Is this two-factor authentication?

[L20]

No. Two-factor requires two **independent** means, and both of these are "what you know." Someone who has compromised your knowledge — by research, phishing or a breach — likely gets both. A password plus a hardware token is two-factor; a password plus another password is not.

4. Why is "cannot be revoked" a more serious objection to biometrics than "false positives"? [L20]

A false positive rate can be tuned by moving the threshold and is a known, bounded cost. Non-revocability is permanent and unbounded: once a biometric template is compromised there is no recovery action available at all, for the rest of that person's life, across every system that uses it. Every other authentication factor has a replacement procedure.

8. Mock quiz

Nine questions, closed-book, 40-minute timer, on paper. Do this before you read §9.

Question 1: Models. Green has a Confidential clearance, Blue has a Top Secret clearance, and `budget.xlsx` is classified Secret.

1.1. Under Bell-LaPadula, who may *read* the file?

1.2. Under Biba, who may *write* the file?

1.3. One of your answers is the same person as the other. Explain why in one sentence.

[L16]

1.1. Blue only. Bell-LaPadula forbids reading up: Green (Confidential) is below Secret, so reading the file would be reading up. Blue (Top Secret) is above it and reading down is permitted.

1.2. Blue only. Biba forbids writing up: Green is below Secret, so writing would be writing up. Blue is above it and writing down is permitted under Biba.

1.3. Both permitted operations are *downward* ones, and Blue is the only subject above the file. Bell-LaPadula permits read-down and Biba permits write-down, so the high subject is the one who qualifies in each case.

Question 2: Model selection. A hospital wants to guarantee that readings written into a patient's chart by an unverified home device never contaminate the official record, while accepting that doctors may read anything. Which model fits, and what does it forbid?

[L16]

Biba, the integrity model. The concern is contamination of trusted data by untrusted data, not disclosure. Biba forbids **write up** — the low-integrity home device cannot write into the high-integrity official record — and forbids **read down**, so a high-integrity process does not ingest low-integrity input. "Doctors may read anything" tells you confidentiality is not the goal, which rules out Bell-LaPadula.

Question 3: Confused deputy. This program is setuid-root:

```
1 int main(int argc, char *argv[]) {
2     FILE *log = fopen("/var/log/audit", "a");
3     FILE *out = fopen(argv[1], "w");
4     /* run the backup, write a report to out, append a summary to log */
5 }
```

3.1. Which line is vulnerable?

3.2. Explain the vulnerability.

3.3. Name the mechanism designed to stop this, and say how it would be applied here.

[L19]

3.1. Line 3.

3.2. The caller controls the pathname through `argv[1]`, but the program opens it with its own root privileges. An unprivileged caller can pass a protected file — `/etc/passwd`, or indeed `/var/log/audit` itself — and the program will truncate it and write the report into it. The program is the deputy: it uses its own ambient authority for an operation directed by a caller who has no such authority. Line 2 is a hardcoded privileged path and is doing its intended job.

3.3. Capabilities. Require a capability granting write access to the caller's chosen output file, obtained against the *caller's* authority, and use a separate capability for appending to the audit log. A pathname alone must not let the caller exercise the program's privilege.

Question 4: Race conditions. A privileged mail program does this:

```
if (access(mailbox, W_OK) == 0) {
    fd = open(mailbox, O_WRONLY|O_APPEND);
    deliver(fd);
}
```

4.1. Name the vulnerability class and mark the check, the window and the use.

4.2. Why was `access()` there in the first place — what was the developer trying to prevent?

4.3. Give the fix in one phrase. [L16 · L19]

4.1. TOCTTOU, a race condition. **Check:** `access` asks whether the *real* user may write `mailbox`. **Window:** between `access` returning and `open` executing, the attacker unlinks `mailbox` and replaces it with a symlink to a protected file. **Use:** `open` re-resolves the name and opens the attacker's target with the program's privileges.

4.2. To avoid a **confused deputy** problem — the developer wanted to confirm the caller could have opened the file themselves before using root authority on it. The intent is right; the implementation separates the check from the use.

4.3. Make the check and use atomic — stop resolving the pathname twice. Open once and work with the returned descriptor (or drop privileges around the open), so there is no window in which the name can be re-pointed.

Question 5: Access control matrix. A lab machine uses DAC. Dana (grader) has read on `answers.txt` and read/write on `notes.txt`. Eli (student) has read on `notes.txt` and no access to `answers.txt`. The matrix enforces this exactly. Describe a way the spirit of the policy is violated anyway, and name what the matrix fundamentally cannot control. [L16]

Dana reads `answers.txt` (permitted) and writes its contents into `notes.txt` (permitted). Eli reads `notes.txt` (permitted) and now has the answers. Every individual access was authorized, and the confidentiality of `answers.txt` is gone. The matrix controls **direct accesses** and cannot control **indirect information flow** — where data goes once a subject legitimately holds it. Controlling that requires a mandatory policy over levels, which is what Bell-LaPadula provides. Note Dana need not be malicious; the same leak occurs by accident.

Question 6: Protecting the OS. A developer proposes that a process be allowed to raise its own segment-descriptor limits, arguing it only ever needs more of its own memory. Explain what goes wrong, and why a system call is the right way to get the same service. [L14]

A malicious process extends its descriptor to cover another process's memory or the kernel's, then reads secrets or corrupts OS code and the protection data itself. The descriptors are the isolation mechanism, so a process that can edit them is not isolated — self-enforced isolation is not enforcement. The descriptors must be writable only by privileged OS code with hardware enforcing them on every access. A **system call** gives the same service safely: control transfers to a designated kernel entry point, the kernel validates the request and its arguments, performs the permitted work, and returns to user mode. The process gets more memory; it never gets kernel privileges.

Question 7: Memory reuse. An OS keeps running processes isolated from each other, and when a process exits it hands the pages to the next process without clearing them. Is isolation sufficient? Name the feature that addresses the gap. [L14]

No. Isolation governs two processes that are running at the same time; it says nothing about data left behind by a previous owner. The new process reads the old secrets from memory it is now *legitimately entitled* to access, so no isolation rule is broken — which is exactly why isolation cannot catch it. The feature is **object sanitization**: zero the memory before exposing it to a different process. Merely freeing it is not sufficient. (This is temporal separation, and temporal separation always requires explicit cleanup at the handover.)

Question 8: Passwords. A site stores **SHA-256(password)** with no salt and no iteration, and rate-limits logins to three per minute per account. It is breached.

8.1. Which guessing attack does the rate limit no longer impede, and why?

8.2. The attacker notices two accounts share a stored hash. What do they learn, and which defense would have prevented it?

8.3. The site adds unique salts afterwards. Is a user whose password is **sunshine** now safe?

[L18]

8.1. Offline guessing. The attacker now runs guesses on their own hardware against the stolen hashes, so the service never sees them and cannot count or refuse them. Only the cost of computing a hash slows them, and plain SHA-256 is very fast.

8.2. That the two users chose the **same password** — hashing is deterministic, so identical inputs give identical outputs. Crack one and you have both. **Unique salts** prevent it, since different salts make the hash inputs different and the stored hashes differ.

8.3. No. The salt is stored alongside the hash, so the attacker takes this user's salt, hashes common passwords with it, and compares. **sunshine** is in every dictionary and falls almost immediately. Salting prevents a single precomputed table from working across many accounts; it does not make a guessable password unpredictable.

Question 9: Search space. A system requires 6-character passwords from the 62 alphanumeric characters. An attacker can test 10^6 guesses per second offline.

9.1. How many possible passwords, roughly, and what power of two?

9.2. How long on average to find one by exhaustive search?

9.3. The real attack succeeds in under a second. Explain the discrepancy. [L18 · L20]

9.1. $62^6 \approx 5.68 \times 10^{10}$. $\log_2(62) \approx 5.95$, so $6 \times 5.95 \approx 35.7$ — call it 2^{36} .

9.2. Average is half the space: $\approx 2.84 \times 10^{10}$ guesses $\div 10^6$ /s $\approx 2.84 \times 10^4$ seconds $\approx$ **8 hours**.

9.3. The calculation assumes passwords are drawn **uniformly at random**, and people do not do that. The attacker tries common passwords, dictionary words and standard substitutions first, searching a space of perhaps a few million rather than 5.7×10^{10} . Exhaustive-search time is an upper bound on the attacker's effort, not an estimate of it.

9. The real practice set, worked

The nine questions from the practice set handed out before the quiz, with worked answers and notes on where marks get lost. If you only have twenty minutes left, read this.

1. Mandatory access control models. Purple has Top Secret clearance, Orange has Unclassified clearance, and the file is classified Secret. (1.1) Under Biba, who may read the file? (1.2) Under Bell-LaPadula, who may write the file? [L16]

1.1. Orange only. Biba prohibits reading down. Treating the levels as integrity levels, Orange may read up from Unclassified to Secret, but Purple may not read down from Top Secret to Secret.

1.2. Orange only. Bell-LaPadula prohibits writing down. Orange may write up to Secret, but Purple may not write down to Secret.

Where marks get lost. Both answers being the same person feels wrong and tempts a second-guess. It isn't: Orange is the low subject, and reading up (Biba) and writing up (Bell-LaPadula) are precisely the two operations a low subject is allowed. Also note the levels are labelled with *confidentiality* words even in the Biba part — treat them as integrity levels and apply the rule mechanically rather than reasoning about who "deserves" access.

2. More mandatory access control. MAC models make use of which security principles? [L16]

Principle of least privilege and complete mediation (the third option). Subjects should receive only the access needed for their work, and every access must be checked against the policy. A reference monitor enforces the policy through complete mediation; the policy must also be configured to grant appropriately limited privileges.

Where marks get lost. "Separation of privilege" is the near-miss distractor — it is a real principle but it concerns requiring two parties for a sensitive action, which is not what levels and a reference monitor do. Anchor on the two mechanisms MAC actually has: levels give least privilege, the reference monitor gives complete mediation.

3. Confused deputy attack. This setuid-root listing is vulnerable:

```
1 FILE *fp1, *fp1;
2 // open argv[2] for writing
3 fp1 = fopen(argv[2], "w");
4
5 // open fstab file to append
6 fp2 = fopen("/etc/fstab", "a");
7
8 /* update the /etc/fstab file, write any error messages
9    to the user-provided file, and exit */
```

(3.1) Identify the vulnerable line. (3.2) Explain the vulnerability. (3.3) What mechanism is designed to stop this? [L19]

3.1. Line 3: `fp1 = fopen(argv[2], "w");`

3.2. The caller controls the pathname, but the program opens it using its root privileges. An unprivileged caller can supply a protected file's pathname, causing the program to truncate that file and possibly write error messages into it. The program is the deputy: it incorrectly uses its own authority for a caller-directed operation.

3.3. Capabilities. Require a capability that grants write access to the caller's chosen output file, and use a separate capability for appending to `/etc/fstab`. A pathname alone must not let the caller exercise the program's privileged authority.

Where marks get lost. Naming line 6 — it touches a protected system file, which looks alarming, but that is the program's intended privileged job and the path is hardcoded. The flaw is always where caller-controlled name meets program-supplied privilege. Also note the declaration on line 1 repeats `fp1`; that is a typo in the listing, not the security flaw, and saying so costs nothing.

4. Protecting the OS. A student suggests letting user programs modify their own base—bounds registers so they can request more memory without a system call. Explain how a malicious program could abuse this, why the OS must control these registers, and how a system call lets a program request a service without unrestricted kernel privileges. [L14]

A malicious process could change its permitted address range to include another process's memory or the kernel's memory, letting it steal secrets or corrupt OS code and protection data. The registers must therefore be controlled by privileged OS code, with hardware enforcing their limits. A system call transfers control to a **designated kernel entry point**; the kernel validates the request and its arguments, performs the permitted work, and returns to user mode. The caller's own code never gains unrestricted kernel privileges.

Where marks get lost. Answering only the first third. The question has three parts and the third — the service-versus-privilege distinction — is the conceptual one. Use the words *designated entry point*, *validates*, and *returns to user mode*; they are what separate a syscall from simply letting the program do it.

5. Memory reuse and isolation. The OS prevents two processes from accessing each other's memory while running. After a process exits, its memory is given to a new process without clearing it. Is process isolation alone sufficient? [L14]

No. Isolation between running processes does not erase data left behind by a previous owner. The new process can read secrets from memory that it is now legitimately allowed to access. **Object sanitization** — zeroing memory before exposing it to a different process — prevents this disclosure. Merely freeing the memory is not sufficient.

Where marks get lost. Not naming the feature. The question says "identify the trusted-system feature," so the phrase **object sanitization** has to appear. The other half worth stating explicitly: no isolation rule was violated, because the new process is genuinely entitled to the memory it now holds — which is why a stronger isolation mechanism would not help.

6. Discretionary access control. Alice (instructor) has read/write on the exam and syllabus; Bob (TA) has read on the exam and read/write on the syllabus; Carol (student) has read on the syllabus only; everyone can print. The matrix enforces this correctly. Describe one way the spirit of the policy could be violated. [L16]

Bob can read the exam and copy its contents into the syllabus, which he may write. Carol can then read the exam contents from the syllabus. Every individual access is permitted by the matrix, yet the exam's confidentiality is lost. The matrix controls direct accesses but does not prevent this **indirect information flow**.

Where marks get lost. Being vague — "information could leak" earns little. Name the specific two-hop path: who reads what, who writes where, who reads the destination. The closing sentence about direct access versus information flow is what connects this to §4 and why MAC exists.

7. Online and offline password guessing. A service allows five failed logins per account per minute. An attacker steals the password database. (7.1) Why does the rate limit slow online but not offline guessing, and how does the attacker check a guess without reversing the hash? (7.2) How does increasing hashing iterations affect the attacker, and what does it cost the service? [L18]

7.1. Online guesses go through the login service, which can enforce the rate limit. Offline guesses run on the attacker's own hardware, so the service cannot count or restrict them. The attacker applies the same password-hashing procedure to each candidate — using the stored salt and parameters if applicable — and compares the result with the stolen hash. This tests guesses without inverting the hash function.

7.2. More iterations increase the computational cost of each guess, reducing the attacker's guessing rate for fixed computing resources. The legitimate service must also do more work for each password verification, increasing login latency and server load.

Where marks get lost. In 7.1, forgetting the second half. "The attacker computes forwards and compares" is the insight the question is testing — a hash is never reversed, it is re-run. In 7.2, forgetting the cost to the service; the answer is a trade-off question and half the marks are in the second clause.

8. What does a salt protect? Two users choose the same password; each is hashed with a different unique salt, stored alongside. (8.1) Should the hashes match, and why do unique salts make a precomputed dictionary less useful even though salts are public? (8.2) A developer claims salting makes a common password safe from offline guessing. Agree? [L18]

8.1. The hashes should generally differ because the salts make the hash inputs different; accidental collisions are possible but unlikely with a suitable hash function. A table computed for one salt cannot generally be reused for another, so the attacker must compute candidate hashes separately for each distinct salt. The salts need not be secret to prevent this reuse of work.

8.2. No. The attacker knows the user's salt and can hash common password guesses with it, comparing each result with the stolen hash. Salting limits precomputation and the sharing of work across accounts; it does not make an easily guessed password unpredictable.

Where marks get lost. Saying salts are secret, or that they "add entropy to the password." They do neither. The whole value is that *work cannot be reused* — one table per salt instead of one table for everyone. State the cost to the attacker in those terms and 8.2 answers itself.

9. Password guessing strategies. (9.1) One attacker uses personal information against a particular user; another tries a few common passwords against thousands of accounts. Which is targeted and which is trawling, and why does a per-account rate limit still allow the second attacker many guesses? (9.2) An attacker uses working username–password pairs from one site's breach on another site. Name the attack and explain why a long random password is still vulnerable if reused. [L18]

9.1. The first is a **targeted** attack; the second is a **trawling** attack. A per-account limit restricts guesses against each account separately. By trying only a few guesses per account, the trawling attacker stays within each account's limit while making many guesses in total across thousands of accounts.

9.2. Credential stuffing. If the same credentials work on another site, the attacker submits the already-known password directly. Its length and randomness do not help once it has been exposed; using a different password for each website prevents this reuse.

Where marks get lost. In 9.1, explaining the limit without explaining the arithmetic — the point is that the attacker's budget is counted *per account* while their goal is satisfied by *any* account. In 9.2, the phrase to include is that this is not guessing at all, so guessing defenses (length, randomness, rate limits) are all beside the point.

10. One-screen cram sheet

Everything the quiz covers, in the order you would want it in the last ten minutes before the room opens.

OS RESPONSIBILITIES. Manage resources (fork, mmap) · interface to hardware (read, write) · user authentication (getuid, setuid) · mediate IPC (pipe) · protect itself (kill, mprotect). All five serve security policies.

SYSCALL. How user space asks the OS for something. Control goes to a **DESIGNATED KERNEL ENTRY POINT.** ; the kernel validates the request, does the work, returns to user mode. The caller gets a **SERVICE.** , never **PRIVILEGES.** .

RUID vs. EUID. RUID = who started the process. EUID = what the OS uses to decide permissions. Normally equal. With the **SETUID BIT.** , EUID becomes the file owner's (usually root) while RUID stays the user's.

MEMORY PROTECTION. Fence (hardcoded OS boundary) → fence register (configurable) → base-bounds (per-process region) → segments (<seg#, offset>, per-region R W X M F, virtual memory). HW + OS *together* protect one process from another.

NEVER LET A PROCESS SET ITS OWN LIMITS. It would widen its range over another process's or the kernel's memory. The registers *are* the isolation; self-enforced isolation is not enforcement.

FOUR SEPARATIONS. Physical (secure, inefficient) · temporal (time sharing; needs sanitization) · logical/algorithmic (address spaces, tabs) · cryptographic (key management). Efficiency ↑, complexity ↑, security ↓.

TRUSTED-SYSTEM FEATURES. TCB (small, well defined) · trusted path (authenticates the OS *to the user*) · secure start-up · object sanitization · auditing.

OBJECT SANITIZATION. Isolation protects *running* processes. It does nothing about freed memory handed to a new owner, who reads old secrets from memory it is legitimately entitled to. Zero it before reuse; freeing is not enough.

RACE CONDITION. Concurrent access is not serializable: 2+ parties share a resource and the final state depends on relative timing. The bug is that one logical operation was implemented as several interleavable ones.

TOCTTOU. check → state changes → use. The decision was true when made and false when relied on. Classic: `access()` then `open()`, with a symlink planted in the gap.

RACE FIX. MAKE CHECK AND USE ATOMIC. — lock, transact, or operate on a handle obtained once instead of re-resolving a name. Not more checks, not stricter permissions, not process ordering.

TOCTTOU VIOLATES... **COMPLETE MEDIATION..** The access that actually happened was never checked — only an earlier, now-stale one was.

ACCESS CONTROL. Which **SUBJECTS.** (users, processes) may access which **OBJECTS.** (files, memory, I/O devices, users, processes) with which **RIGHTS.** (read, write, execute, create, transfer).

THREE BEST PRACTICES. **UNIVERSAL APPLICATION.** (check every access) · **LEAST PRIVILEGE.** (minimum needed, supports failing safe) · **TYPE CHECKING.** (operations must be meaningful for the object).

WEAKENED MEDIATION. Random checking · random auditing (*detection, not prevention*) · selective checking. All are compromises on universal application.

DAC STORAGE. **MATRIX.** (easy, easy revocation, but sparse) · **DIRECTORY.** per subject (fixes sparsity; file revocation hard, aliasing) · **ACL.** per object (file revocation easy, no aliasing; user revocation hard).

WHAT DAC CAN'T DO. Bob reads the exam, writes the syllabus; Carol reads the syllabus. Every access authorized, confidentiality gone. Matrices control **DIRECT ACCESS.** , not **INDIRECT INFORMATION FLOW.** . That gap is why MAC exists.

POLICY FAMILIES. Discretionary (owner decides) · Role-Based · Attribute-Based · Mandatory (system-wide, owner cannot override).

MAC. Goal: understand and control how information *flows*. Multi-level security + reference monitor. Rests on **LEAST PRIVILEGE + COMPLETE MEDIATION.** .

LEVELS. **CLEARANCE.** = a subject's level. **CLASSIFICATION.** = an object's level. top secret → secret → confidential → restricted → unclassified.

BELL-LAPADULA. Confidentiality. **NO READ UP, NO WRITE DOWN..** You can't see above you or spill below you. Information flows *upward* only.

BIBA. Integrity. **NO WRITE UP, NO READ DOWN..** You can't contaminate above you or drink from below. Information flows *downward* only. Exactly the mirror of BLP.

WORKED CASE. Purple = Top Secret, Orange = Unclassified, file = Secret. Biba read? **ORANGE ONLY..** BLP write? **ORANGE ONLY..** Both permitted ops are upward, and Orange is the low subject.

READ/WRITE AT OWN LEVEL ONLY. Satisfies both models' rules → provides **CONFIDENTIALITY AND INTEGRITY.** , but not availability.

REFERENCE MONITOR. Enforces the policy. Must be **COMPLETE MEDIATION** · **TAMPERPROOF** · **VERIFIABLE**. . Part of the TCB. Real-world example: a network firewall.

ANALOG HOLE. Purple reads the Secret file and just *tells* Orange. No system rule broken. A model governs only the channels it mediates.

AMBIENT AUTHORITY. All the extant authority of the current execution context. Authority accrues as a program runs, so a setuid program holds several permission sets at once and nothing says which to use.

CONFUSED DEPUTY. A program with multiple permission sets uses its ambient authority indiscriminately. Deputy = the privileged program · attacker = the unprivileged caller · confusion = it doesn't realise it's acting on someone's behalf.

SPOT THE LINE. Always where **THE CALLER CONTROLS THE PATHNAME.** (`argv[...]`) **AND THE PROGRAM SUPPLIES THE PRIVILEGE.** . A hardcoded privileged path is not the bug.

ACCESS CONTROL CAN'T FIX IT. The kernel sees a privileged process asking for a file root may write. The check passes correctly. A **PATHNAME CARRIES NO AUTHORITY.** , so nothing records that an unprivileged user chose it.

CAPABILITIES. Object reference + access right — a "fat pointer": where the object is AND what you may do (read, write, execute, grant). **UNFORGEABLE.** · ties right to object · **NO DESIGNATION WITHOUT AUTHORITY.** . Gives no ambient authority, supports least privilege.

DEPUTY VS. TOCTTOU. Deputy = the check *never happens*, no timing needed. TOCTTOU = the check happens and is *correct*, then the state changes. Fixes: capabilities vs. atomicity.

SEARCH SPACE. 94 printable non-space chars. $94^8 \approx 6.1 \times 10^{15} \approx 2^{53}$. At $10^4/s$, average (half) $\approx$ **14,300 YEARS.** . 2^{56} key at $10^{10}/s \approx 83$ days; at $10^{15}/s \approx 72$ s.

THE THREE STEPS. Alphabet^{length} $\rightarrow$ **HALVE IT.** (average case) $\rightarrow$ divide by guess rate. Powers of two: length $\times \log_2(\text{alphabet})$; $\log_2(94) \approx 6.55$.

TARGETED VS. TRAWLING. **TARGETED.** = one specific user; common passwords then personal info; countered by rate limiting. **TRAWLING.** = access through *any* user; a few guesses on thousands of accounts stays inside every per-account limit.

CREDENTIAL STUFFING. Breached pairs replayed on another site. Not guessing at all — so length, randomness and rate limits are all irrelevant. Only unique passwords per site help.

ONLINE VS. OFFLINE. Online goes through the service, which can count and refuse. Offline runs on the attacker's hardware — invisible, uncountable, unlimitable. Only hash cost slows it.

HOW OFFLINE GUESSING WORKS. Hash the candidate *forwards* with the stored salt and parameters, compare with the stolen hash. The hash is never inverted — the attacker just runs the service's own verification procedure.

PRECOMPUTED DICTIONARY. Hashing is deterministic, so hash likely passwords once and look up every stolen hash. Same password $\rightarrow$ same hash, visible in the file.

ITERATED HASHING. d iterations $\rightarrow$ each guess costs $d\times$ more, attacker's rate drops $d\times$. The service pays the same $d\times$ on **EVERY LEGITIMATE LOGIN.** : more latency, more load.

SALT. A per-user string concatenated before hashing. **PUBLIC, NOT SECRET..** Does: kills reuse of precomputed work — one table per salt, not one for everyone. Does NOT: make a guessable password safe.

WHY THE MATH LIES. Passwords aren't uniform. Common passwords (tiny set) $\cdot$ names $\cdot$ $\sim 600,000$ dictionary words $\cdot$ substitutions (e:3, o:0, t:7, l:1, for:4). Exhaustive-search time is an upper bound, not an estimate.

ID VS. AUTH. IDENTIFICATION. asserts who you are — public, well known.

AUTHENTICATION. proves it — private, unforgeable, authentic. A name tag identifies and authenticates nothing.

FOUR MEANS. What you know (passwords) $\cdot$ what you do/are (biometrics) $\cdot$ what you have (tokens) $\cdot$ where you are. **2FA.** = two *independent* means.

BIOMETRICS. FTE (can't enrol) $\cdot$ FTC (can't capture) $\cdot$ false positives/negatives traded by the **THRESHOLD.** in the overlap of two score distributions. Bob's area above the threshold = chance Bob authenticates as Alice.

BIOMETRIC CONS. Insecure remote or unsupervised $\cdot$ **CANNOT BE REVOKED.** $\cdot$ false positives $\cdot$ privacy. Security is lower than people expect.

Built October 9, 2026 from the L14–L20 lecture decks, the in-class practice sheets and the Quiz 3 practice set of COMP 435 (Fall 2026, UNC). Personal study material.

© 2026 Deven Jadhav. Not affiliated with or endorsed by UNC-Chapel Hill; course material © its respective owners. Willful large-scale copyright infringement may be a federal crime (17 U.S.C. §506).