COMPSCI 750 – Final Exam
Gunnar Yonker

**1.**

**(1)**

**Access Control List (ACL)**

An ACL is a list that specifies which users or system processes are granted access to objects and what operations are allowed on given objects. Each entry in an ACL specifies a subject and an operation. For example, an ACL for a file might include a list of users and their corresponding permissions (read, write, execute).

ACLs are commonly used in file systems, databases, and network protocols to define permissions for resources. They are more appropriate in scenarios where the permissions are often tied to the object/resource itself, like files in a file system or rows in a database.

**Capability-Based Access Control**

Capability-based access control centers around the idea of capabilities – tokens of authority that a subject can use to access objects. Rather than checking if an object's ACL allows a certain subject to perform an operation, the system checks if the subject has a capability that allows them to perform the operation on the object.

Capability systems are often seen in certain operating system designs or specialized systems where the permissions need to be handed out in a more decentralized fashion, or where it's beneficial to have subjects prove their rights without referring back to the central authority (the object) frequently.

**(2)**

**Bell-LaPadula Model**

Main Ideas: This is a formal security model focused on maintaining data confidentiality in computer security systems. It introduces the concepts of security labels and mandatory access controls. The Bell-LaPadula model is essentially a state machine with well-defined state transitions that ensure that every state transition maintains the "secure state" of the system. The primary properties include:

Simple Security Property (no read up, or "ss-property"): A subject with a lower security clearance cannot read data at a higher clearance.

Star Property (no write down, or "*-property"): A subject with a higher security clearance cannot write to an object with a lower clearance.

Connection with Information Flow Control: Both models aim to control the flow of information in a system to ensure security. Bell-LaPadula focuses on preventing information from flowing from a higher classification level to a lower one, which is a type of information flow control.

Major Differences: Information flow control models can address a wider array of security concerns, including both confidentiality and integrity, whereas Bell-LaPadula is primarily focused on confidentiality.

COMPSCI 750 – Final Exam
Gunnar Yonker

**(3)**

**Low Water Mark Policy**

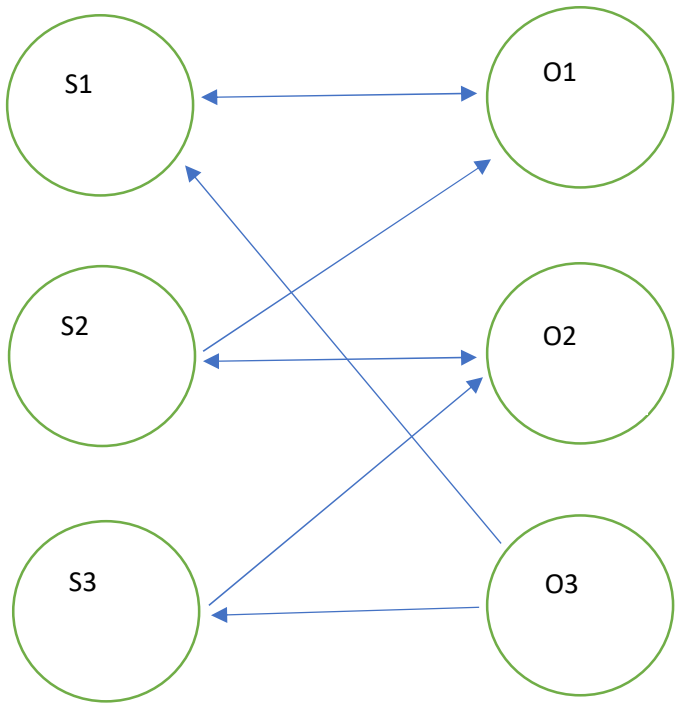
Main Ideas: The Low Water Mark policy is concerned with the integrity of information. In some contexts, it can also address confidentiality issues (especially when considering data aggregation in multi-level secure systems). When information from various sources of different integrity levels is combined, the resulting integrity level is set to the lowest (or "worst") integrity level of any of the sources. This ensures that high-integrity data is not mistakenly elevated due to its association with lower-integrity data.

Integrity Assurance: While the Low Water Mark policy helps in preventing high-integrity data from being elevated inappropriately, it doesn't necessarily ensure the overall integrity of a system. The policy is conservative, meaning that it might often downgrade the integrity of data more than necessary. As a result, it can lead to scenarios where high-integrity data is treated as suspect merely because it was combined with low-integrity data. For true integrity assurance, more comprehensive policies and measures are required.

**2. Information Flow Model**

| | O1 | O2 | O3 |
|---|---|---|---|
| S1 | RW | -- | R- |
| S2 | -W | RW | -- |
| S3 | -- | -W | R- |

**(1)** Information Flow Graph



COMPSCI 750 – Final Exam
Gunnar Yonker

**(2)** Possible Secrecy Leaks

Based on the flow graph:

- O3's information can flow to both S1 and S2. If S1 writes to O1 or O2, then S2 or S3 can indirectly get information from O3.
- O2's information can flow to S2. If S2 writes to O1, then S1 can indirectly get information from O2.
- O1's information can flow to S1. But since nobody else can read from O1 directly or indirectly, there's no further leak from O1.

**(3)** Integrity Levels

High Integrity: S1 has the highest integrity as it does not rely on any inputs from other subjects.

Low Integrity: S2 and S3 can be considered to have low integrity. S2 reads from O2, which S3 can write to, making S2's integrity questionable. Similarly, S3 reads from O3, which S1 can write to, making S3's integrity questionable as well.

**(4)** Applying the "no read up, no write down" Principle

S1: Top Secret

S2: Secret

S3: Confidential

O1: Top Secret

O2: Secret

O3: Confidential

Subjects at lower levels cannot read from higher level objects:

S2 should not read from O1.

S3 should not read from O1 and O2.

Subjects at higher levels cannot write to lower level objects:

S1 should not write to O2 and O3.

S2 should not write to O3.

COMPSCI 750 – Final Exam
Gunnar Yonker

Updated Matrix:

| | O1 (Top Secret) | O2 (Secret) | O3 (Confidential) |
|---|---|---|---|
| S1(Top Secret) | RW | -- | R- |
| S2 (Secret) | -W | RW | -- |
| S3 (Confidential) | -- | -W | R- |

It looks like the original table already follows the “no read up, no write down” principle, but here is an analysis to double check.

S1 can read and write to 01, same security level. S1 can read from O3, which is reading down and is allowed.

S2 can write to O1, write up which is allowed. S2 can read and write to O2, same security level.

S3 can write to O2, write up which is allowed. S3 can read from O3, same security level.

**3. Security Design Principles**

**(1)**

Least Privilege:

Every module (such as a process, a user, or a program, depending on the subject) must be able to access only the information and resources that are necessary for its legitimate purpose. By minimizing the access rights of a module, the potential damage or misuse in case of errors or compromises is limited.

Fail-safe Defaults:

The default disposition of access should be denial unless access is explicitly granted. Unless a user or process has been given specific permission to access a resource, they should be denied. This ensures that if an oversight occurs, the default state is a secure one.

Economy of Mechanism:

Security mechanisms should be as simple as possible. Simpler designs are easier to test and verify for correctness. Complex mechanisms can hide security vulnerabilities.

Complete Mediation:

Every access to every resource must be checked for authority. This prevents bypassing of security checks. It’s crucial for ensuring that unauthorized actions cannot occur without detection.

Open Design:

The design should not be secret, and the mechanisms should not depend on the ignorance of potential attackers to remain secure. Security by obscurity is not a robust strategy. A system’s security should be verifiable independent of the secrecy of its design.

COMPSCI 750 – Final Exam
Gunnar Yonker

Separation of Privilege:

The system should not grant permission based on a single condition. Multiple conditions reduce the risk of a security breach by requiring an attacker to penetrate more than one safeguard.

Least Common Mechanism:

Minimize the amount of mechanisms (like libraries or utilities) shared by different users. Sharing can introduce channels or unintended information flow or breaches.

Psychological Acceptability:

The human interface must be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Security mechanisms that confuse or frustrate users can inadvertently lead to vulnerabilities if users seek ways around them or misuse them.

**(2)**

Least Privilege:

The machines ran with elevated privileges, which allowed them to perform a wide range of tasks beyond what was necessary for voting. Such an unrestricted environment poses a significant risk if the machine becomes compromised. Malicious software can modify all records, audit logs, and counters. The study demonstrated that a vote-stealing attack can occur without detection. The system does not adhere well to the principle of least privilege. Many parts of the software had broader access than was necessary, increasing the attack surface and potential for malicious behavior. A system designed with the principle of least privilege in mind would segregate duties and restrict the ability of any single component or process to modify crucial records without checks and balances.

Open Design:

The design and functioning of the Diebold voting machines were proprietary and not open to the public for inspection. This secrecy raised suspicions and made independent verification of the system's security difficult. The system violates the open design principle. The machine operated on version 4.3.15 of Diebold BallotStation software, which had known vulnerabilities from a prior leak. The conclusion that subsequent versions of the software should be assumed insecure until independently verified suggests a lack of transparency. The reliance on security through obscurity, instead of openly verifiable and transparent security mechanism, weakens the overall trust in the system. An open design would allow for broader inspection and more rapid discovery and correction of flaws.

Psychological Acceptability:

The machine's user interface was relatively straight forward for voters. However, election officials and other individuals responsible for setting up, programming, and auditing the machines may have had challenges due to the machine's complexities and non-intuitive security mechanisms. Some of the detected vulnerabilities could be exploited simply by using functionalities available in the standard interface, making it challenging for operators to distinguish between standard and potentially malicious behavior. Anyone with brief physical access can install malicious software. The real-world implications of this finding mean that polling staff or any individual with unsupervised access to the machines could unknowingly or intentionally compromise the system. From a voter's perspective, the system may seem

acceptable because the interface is straightforward. However, from a security standpoint, the system's vulnerabilities could be masked by its simplicity, leading to a false sense of security. The system partially adheres to this principle but fails to account for the deeper implications of psychological acceptability, which should also consider the confidence and trust that both operators and voters have in the system's integrity.

**4.**

**(1)** Bell-LaPadula Model

| Subject/Document | Doc1 (SECRET, {A}) | Doc2 (TOP SECRET, {A,B,C}) | Doc3 (UNCLASSIFIED, {B}) |
|---|---|---|---|
| S1 (TOP SECRET, {A,C}) | Read, Write | Read, Write | Read, No Write |
| S2 (SECRET, {A}) | Read, Write | No Access | Read, No Write |
| S3 (CONFIDENTIAL, {B,C}) | No Access | No Access | Read, No Write |

**(2)** Biba Model:

| Subject/Document | Doc1 (SECRET, {A}) | Doc2 (TOP SECRET, {A,B,C}) | Doc3 (UNCLASSIFIED, {B}) |
|---|---|---|---|
| S1 (TOP SECRET, {A,C}) | No Access | Read, Write | No Access |
| S2 (SECRET, {A}) | Read, Write | Read, No Write | Read, No Write |
| S3 (CONFIDENTIAL, {B,C}) | Read, Write | Read, No Write | Read, Write |

**5. Buffer Overflow Attack**

**(1)**

The buffer is 24 bytes in size.

The buffer starts at address 0xAABB0010.

The return address is stored at 0xAABB0050.

0xAABB0050 – 0xAABB0010 = 0x40 = 64 bytes

To overwrite the return address, the string provided must have a length of 64 bytes + 4 bytes. The first 64 bytes will fill up the buffer and the space between the buffer and the return address, and the next 4 bytes will overwrite the return address itself.

0-23 bytes: Actual buffer space.

24-63 bytes: Padding, to reach up to the return address.

64-67 bytes: Overwrite return address to point to the location of the malicious code.

String constraints will be organized like this:

| buffer[24] | padding[40] | return address[4] | payload |

Buffer[24] is any 24 bytes, typically filled with NOPs or random data.

COMPSCI 750 – Final Exam
Gunnar Yonker

padding[40] is the 40 bytes needed to reach up to the return address.

return address[4] is the address where the malicious code/payload starts.

payload is the injected code.

Example String:

char exploit_string[] =

"AAAAAAAAAAAAAAAAAAAAAAAA"  // 24 bytes - buffer

"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"  // 40 bytes - padding

"\x10\x00\xBB\xAA";  // 4 bytes - address of the payload, pointing to the beginning of buffer for this example (little endian format) – works if planning to place shellcode at the start of the buffer and using a NOP sled

Thus:

"AAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\x10\x00\xBB\xAA"

**(2)**

Stack Canaries: These are random values placed on the stack before local variables. Before a function returns, the system checks whether the canary value has changed. If it has, the program aborts, preventing a potential exploit from executing.

Address Space Layout Randomization (ASLR): This technique randomizes the memory addresses used by processes, making it difficult for attackers to predict the exact location to jump to for their malicious code.

Non-Executable Stack: This defense technique ensures that the stack memory region is non-executable. Hence, even if an attack injects malicious code onto the stack, the system won't execute it. This technique can be combined with Data Execution Prevention (DEP). However, this can be bypassed with techniques like return-oriented programming (ROP).

Static and Dynamic Analysis: Using static analysis tools to analyze the codebase for potential vulnerabilities before runtime, and dynamic analysis tools during runtime to detect any anomalies or suspicious activities. This analysis can help identify and prevent potential exploits before they become an issue.

Code Auditing: Regularly audit and review the codebase to identify and eliminate potential buffer overflow vulnerabilities. It's essential to educate developers about secure coding practices.

COMPSCI 750 – Final Exam
Gunnar Yonker

**6. Race Condition**

**(1)**

The given code snippet does have a race condition vulnerability known as a Time-of-Check to Time-of-Use (TOCTTOU) race condition. This vulnerability exists because there is a time gap between when the program checks for the permissions of the file (using the access function) and when it actually opens the file (using the open system call). The attack window is the time between the check (access call) and the use (open call). An attacker can exploit this window by replacing the file /tmp/XYZ after the permission check but before the file is opened by the program.

To exploit this vulnerability, an attacker could use a technique called symbolic linking. The attacker waits for the program to perform the access check and then immediately after the check, but before the open call, the attacker replaces /tmp/XYZ with a symbolic link pointing to a sensitive file they normally wouldn't have permission to write to. The program, running with elevated privileges (due to Set-UID), will then open and write to this sensitive file, thinking it's writing to /tmp/XYZ. The attack can automate this process to continuously attempt to create this symbolic link so that it runs continuously until the exploit is successful. This exploit relies on replacing the symbolic link in the right window of time, which is why continuously running an exploit program will eventually find that window of time to successfully write to the intended file. For example, trying to add a new user with a default password (or no password as we saw in our labs) that has root access on a system to the /etc/passwd file (which requires elevated permissions to access).

**(2)**

Yes, the open system call as described exhibits a check-and-then-use pattern, which is a common source of TOCTTOU race conditions. The race condition arises because the state of the system might change between the check and the use. In this case, between the permission check and the actual file open operation. This can lead to unintended behavior, such as writing to a file that the program shouldn't have access to. To fix this issue, the program should avoid separate check and use operations. Instead, it should attempt the operation directly and handle any failures (e.g., permission errors) that arise. For example, instead of checking for wr4ite permissions and then opening the file, the program should just attempt to open the file for writing and handle any errors that occur if the file is not writable.

COMPSCI 750 – Final Exam
Gunnar Yonker

**7.**

**(1)**

For data segment d with access bracket (1,4):

p is denied for accessing:

A process would be denied access if it executes outside of the range specified in the access bracket. In this case, that means:

Ring 0

Rings 5 through 7

p is allowed to read, write, or execute:

Given the ACL provides full access rights (read, write, execute) to process p, the process would be able to perform these operations when it's executing in the range specified by the access bracket. So:

Rings 1 through 4

p is allowed to read but not write or execute:

For the process to be able to read but not write or execute, it must be executing in a ring that is outside the access bracket. Any ring outside of this range would be denied all access, not just write or execute. So, no rings would fit this statement.

**(2)**

For processes p1 and p2, where p2 has an access bracket of (4,5) and a call bracket of (5,6):

p1 cannot invoke:

For p1 to be unable to invoke p2, it would have to be executing outside of both p2's access and call brackets. This means:

Rings 0 through 3

Ring 7

p1 can invoke p2 but a ring-crossing fault occurs:

This would happen when p1 is operating in a ring that's inside p2's call bracket but outside of its access bracket. Thus:

Ring 6

COMPSCI 750 – Final Exam
Gunnar Yonker

p1 can invoke p2 if a valid gate is used as an entry point. A ring-crossing fault occurs:

Using a gate p1 can jump to a ring within p2's call bracket but not to one within its access bracket. Thus, it would still have to be in:

Ring 6

P1 can invoke p2 and no ring-crossing fault occurs, no gate is needed:

This would occur when p1 is executing within the access bracket of p2 because no faults occur in this scenario, and no gate is required. So p1 would need to be in:

Rings 4 and 5