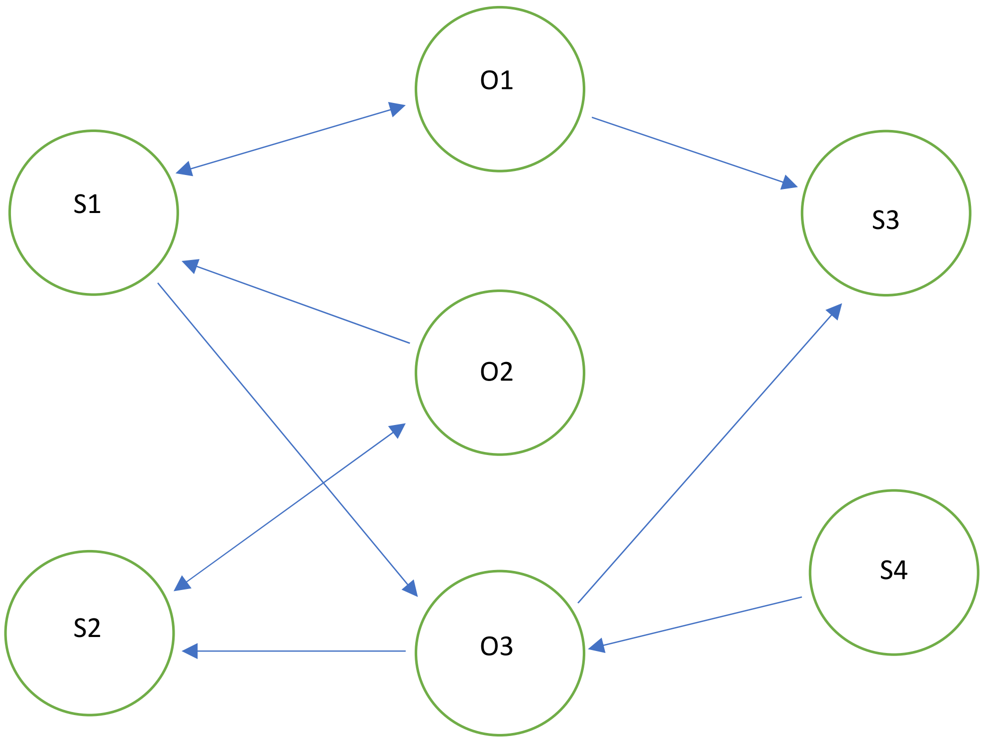
COMPSCI 750 – Assignment 2
Gunnar Yonker

I. Information flow model and Mandatory Access Control Policies

	O1	O2	O3
S1	RW	R	W
S2		RW	R
S3	R		R
S4			W

1.



COMPSCI 750 – Assignment 2
Gunnar Yonker

2.

If there exists a path in the information flow graph from the object (o) to the subject (s), then there is an unauthorized secrecy leak in the corresponding protection state. Using this information, we can see then that there are a few possible secrecy leaks:

Secrets from O1 can potentially reach O3 via S1.

Secrets from O2 can potentially reach O1 or O3 via S1.

Secrets from O3 can potentially reach O2 via S2.

Secrets from S1, S3, or S4 can mix in O3.

Secrets from S1 or S2 can mix in O2.

3.

A process is considered to be of high integrity if it doesn't depend on any low integrity inputs. A subject that reads from an object is depending on that object for information. Looking at our information flow model we can determine what subjects have low integrity and high integrity.

S1 reads from O2, and writes to O1 and O3. Since it reads from O2, it might depend on potentially low integrity data.

S2 reads from O3 and writes to O2. Again, it reads and might depend on potentially low integrity data.

S3 reads from O1, potentially depending on its data.

S4 only writes to O3 and doesn't read from any object, making it high integrity.

From this analysis we can gather that S1,S2, and S3 are low integrity because they depend on inputs from other objects, their actual integrity would further depend on the integrity of the objects they read from. S4 likely has high integrity since it doesn't depend on any inputs.

4.

To ensure that the MLS model follows the principles of "no read up, no write down", we can look for any violations to that principle and then remove those rights. In the no read up part of the principle, S3 (Confidential) reads from O1 (Top Secret) which is a violation. In the no write down part of the principle, S1 (Top Secret) writes to O3 (Confidential) which is a violation. The access rights that need to be removed are as follows:

Remove the read access of S3 from O1

Remove the write access of S1 to O3

Updated access control matrix:

	O1 (Top Secret)	O2 (Secret)	O3 (Confidential)
S1 (Top Secret)	RW	R	
S2 (Secret)		RW	R
S3 (Confidential)			R
S4 (Unclassified)			W

5.

The Biba integrity model follows the “no read down, no write up” principles, so again we can look at each of the access rights to evaluate them based on the Biba model looking for any violations. If any violations are found, the access rights can be removed to make sure this information flow graph adheres to the Biba integrity model.

No Read Down:

S1 (Highest) reads O2 (Second Highest): Violation

S2 (Second Highest) reads O3 (Third Highest): Violation

No Write Up:

S4 (Lowest) writes to O3 (Third Highest): Violation

Access rights to remove to ensure that the Biba integrity model is enforced:

S1's read access to O2

S2's read access to O3

S4's write access to O3

Updated access control matrix:

	O1 (Highest)	O2 (Second Highest)	O3 (Third Highest)
S1 (Highest)	RW		W
S2 (Second Highest)		RW	
S3 (Third Highest)	R		R
S4 (Lowest)			

COMPSCI 750 – Assignment 2
Gunnar Yonker

II. setUID and Environment Variables

1.

Set-UID programs are designed to perform specific tasks with elevated privileges. The program is written such that the root privileges are only used for the specific task it's designed for and not anything else. Hence, the user cannot perform arbitrary actions with those privileges. Additionally, there are security measures in operating systems that mitigate the risks associated with set-UID, such as randomizing memory addresses to prevent buffer overflow attacks. The principle of least privilege ensures that the program only has the necessary permissions to execute its tasks and nothing more. Unsafe behaviors such as mixing code and data or capability leaking can be mitigated by careful programming and employing best practices. Lastly, restrictions on certain system calls or functions can be enforced, especially in set-UID programs to reduce potential exploits.

2.

In this situation, yes the program will be able to read the file. The real user ID would refer to the user who launched the program which is Alice. When a set-UID program is executed, the effective user ID is changed to that of the file's owner (Bob), but the real user ID remains unchanged (Alice). Given that /tmp/x is readable by Alice, the program's real user ID still has the permission to read the file, even though its effective user ID is set to Bob. File access checks first look at the effective user ID, but if the access is denied based on that, then the system checks the real user ID. So, since the real user ID (Alice) has read permissions for /tmp/x, the set-UID program can read the file even though it's running with Bob's effective user ID.

3.

No, the process cannot open the file. Access control is based on the Effective User ID (EUID). This case, the EUID is 1000, which does not have the permission to read the file. The Real User ID (RUID) is not used for access control checks. When a process tries to open a file, it's the effective user ID that determines access rights, not the real user ID. Unlike a set-UID program, standard file access control only considers the EUID. The process will be denied access to the file based on its EUID not having read permission, even though its RUID would have been granted access.

4.

The redirection (<) is managed by the shell, which operates under the user's privileges, not the programs. So, even if prog is a set-UID program, if the user doesn't have permission to read /etc/shadow, the shell won't be able to open that file for redirection. Therefore, this method cannot be used to get the set-UID program to read from the /etc/shadow file unless the user already has permission to read that file. Essentially, the redirection fails at the shell level before the privileged program starts. When you try to redirect from /etc/shadow you are asking the shell (which runs with your user privileges) to open the file. This is before the set-UID program gets a chance to do anything. Since a regular user doesn't have read permissions on /etc/shadow, this operation will fail.

5.

When other users are given execution permissions to this set-UID binary, they will be able to execute it with the effective permissions of the file's owner, which is root. The more command is primarily used to view files in a paginated manner, but it has other capabilities that could pose a security risk when misused with root permissions:

Shell Escape: When viewing a file with "more", a user can press "!" followed by a shell command. This command will be executed in a subshell. If "more" is running as root, then this shell command would also run as root.

Editing Files: While viewing a file, pressing "v" will open the current file in an editor, typically vi or vim, whichever editor is set as the VISUAL or EDITOR environment variable. Since "more" would be running as root in this case, any edits made in the editor would also be made with root permissions. In addition, from editors like vi or vim, it is possible to execute shell commands which again would run with root permissions.

Opening Other Files: From within "more", pressing ":e" followed by a file name will open that file for viewing. Given root permissions, this can be used to read any file on the system.

Considering the above functionalities, there are some clear risks in setting the set-UID bit for "more". Alice can easily gain unintended root-level access to perform a variety of tasks beyond just reading files. This poses a considerable security threat.

6.

When invoking external commands, system() works by calling /bin/sh first. The system() function relies on the PATH environment variable to locate a command if a full path isn't provided. This can pose a risk as an attacker could manipulate the PATH variable, leading the system to execute a malicious command in place of the intended one. Capability leaking is another area of concern, especially for set-UID programs. If a set-UID program uses system() and doesn't clean up or downgrade its privileges properly before invoking a shell command, it can lead to capability leaking. Attackers can exploit this to run arbitrary commands with escalated privileges. Using system(), there is a risk of mixing code and data if not used carefully. For instance, user input can be unintentionally treated as part of a command, leading to command injection attacks.

There are a few reasons why execve() is safer. It directly executes a program without invoking a shell. This means it's not influenced by shell-specific environment variables. With execve(), code and data are clearly separated. This separation ensures that user-provided data can't easily be interpreted as code. It also does not have the capability leaking problems related to shell invocation as system() does. In conclusion, system() has broader attack surfaces mainly because it invoke the shell, making it susceptible to environment variable manipulation and command injection attacks. On the other hand, execve() provides a more controlled way of executing external programs, making it a safe choice, especially in privileged contexts.

7.

getenv() fetches the value of an environment variable. The environment in which a program runs can be influenced by the user. For a regular program, this isn't a concern. However, for example in a set-UID program there is a potential security vulnerability. A malicious user could modify environment variables to influence the behavior of these programs, potentially leading to privilege escalation or other unwanted behaviors. For example, if a set-UID program uses getenv() to fetch the value of the PATH environment variable and then uses this value to execute another program or command, a malicious user could set the PATH to point to a directory containing a malicious version of the desired command, thus executing arbitrary code with elevated permissions.

secure getenv() is designed to mitigate the risks with set-UID programs. It behaves like getenv(), but returns NULL if the program is being run with elevated privileges (like if it's being run as a set-UID program). This ensures that the potentially unsafe environment variables are not inadvertently used in a security-sensitive context. Essentially, using secure getenv() helps maintain the principle of least privilege. Even if a program is running with elevated privileges, it won't be able to fetch environment variables that could be manipulated by the user to perform malicious activities. This makes it the safer choice by reducing the potential attack surface and ensuring that the program doesn't fetch and act upon environment variables that could have been maliciously set by a user.

8.

The PWD environment variables contains the working directory. This variable is convenient and widely used, but there is a major drawback in a privileged context: the environment can be manipulated by users. Relying on PWD in a set-UID program can be dangerous. An attacker might set the PWD variable to a misleading value, potentially causing the program to operate on the wrong directory. The getcwd() function retrieves the absolute pathname of the current working directory. It works by navigating the filesystem hierarchy from the current directory back to the root, constructing the absolute path along the way. This method doesn't rely on potentially tainted environment variables. Based on these observations, I would use the getcwd() approach over relying on the PWD environment variable. By using getcwd(), it is a more trustworthy means of determining the current directory and reduces the risk associated with potential environment variable tampering.