1.

Lets start by defining **Security Vulnerability** and **Exploit**.

**a. Security Vulnerability:**

* **Definition**: A vulnerability is a flaw or weakness in a system, software, hardware, or process that can be potentially exploited by attackers to compromise the system’s security.
* **Examples**:
  + Buffer overflows.
  + SQL injection vulnerabilities.
  + Misconfigured security settings.
  + Outdated software with unpatched bugs.
* **Nature**: Vulnerabilities are passive and represent potential risks. They exist due to coding errors, design flaws, or configuration mistakes but, by themselves, do not cause harm unless exploited.

**b. Exploit:**

* **Definition**: An exploit is an actual attack or technique that takes advantage of a vulnerability to cause harm. It is the action taken by an attacker to leverage a vulnerability to gain unauthorized access, steal data, or perform malicious actions.
* **Examples**:
  + A crafted input that triggers a buffer overflow vulnerability to execute arbitrary code.
  + A script that performs an SQL injection to retrieve sensitive information from a database.
  + Malware that uses a known vulnerability to gain root access.
* **Nature**: Exploits are active; they involve specific steps or tools to take advantage of a vulnerability.

**Key Differences:**

* **Vulnerability**: A weakness or flaw in a system (a potential problem).
* **Exploit**: The method or attack used to take advantage of that weakness (the actual attack).

In short, vulnerabilities are the weaknesses in a system, while exploits are the tools or methods used to attack those weaknesses.

2.

int fact(int x){

int y;

y = x \* ((x>1)? fact(x-1):1);

return y;

}

int main(int argc, char \*\*argv){

return fact(N);

}

(a) This program is compiled for an x86 32-bit Intel processor as follows:

gcc -fstack-protector -O0 -o fact.c

**i. What is the purpose of -fstack-protector and -O0?**

1. **-fstack-protector**:
   * **Purpose**: This option enables **stack protection** against **buffer overflow** attacks by inserting a **canary** (a known value) on the stack before local variables. The canary is checked before returning from a function. If the canary value has been changed (which indicates a buffer overflow), the program is terminated to prevent exploitation.
   * **Usage**: It helps prevent attacks where malicious code tries to overwrite the return address or control flow using buffer overflows.
2. **-O0**:
   * **Purpose**: The -O0 flag disables compiler optimizations. With this option, the compiler generates code without any optimization for performance, resulting in a more straightforward translation from the source code to machine code.
   * **Usage**: This is useful during **debugging** or **educational purposes** since it ensures that the generated assembly closely matches the original C source code. It makes it easier to trace the program behavior, as the code remains unoptimized and easier to follow in debuggers.

**ii. What is the full form of ELF?**

* **ELF** stands for **Executable and Linkable Format**.
  + **Description**: ELF is a common file format used for **executables**, **object files**, **shared libraries**, and **core dumps** in Unix-like operating systems such as Linux. It provides the structure that helps the operating system load, link, and execute the program.
  + **Key Components**: ELF files contain sections such as headers, data, code, and debugging information that the system uses to execute the binary or link it with other binaries.

(b) During execution, the stack and code segments are set to 0x2000 and 0x1000 respectively. Each segment is of 4KB.

**i. Are 0x1000 and 0x2000 virtual addresses or physical addresses?**

* These are **virtual addresses**.
  + In modern operating systems, programs do not directly access physical memory. Instead, they use **virtual memory** addresses, which are mapped to physical memory by the operating system's **memory management unit (MMU)**.
  + So, 0x1000 (code segment) and 0x2000 (stack segment) are virtual addresses. The OS translates these to physical addresses when the program runs.

**ii. What is the initial value of %esp?**

* The initial value of %esp (the **stack pointer**) is the **top of the stack**.
  + In this case, the stack segment starts at virtual address 0x2000 and grows **downwards**.
  + Since the stack size is 4KB, the initial value of %esp will be **0x2000 + 4KB = 0x3000** (the end of the stack segment).
  + So, the initial value of %esp is **0x3000**.

**iii. Suppose the stack frame for main and all functions that execute before it occupy 256 bytes. What is the maximum value of N for which the program works correctly?**

* **Stack Usage**: The program uses recursion to compute the factorial, and each recursive call to fact() creates a new stack frame.
* **Stack Size**: The stack starts at 0x3000 and grows downwards. The stack segment is 4KB (4096 bytes) in size, so the total available stack space is 4096 bytes.
* **Initial Occupied Stack**: Before any calls to fact(), 256 bytes of the stack are already occupied by main and any other pre-existing functions. Therefore, the available stack space for the recursive fact() calls is:

4096−256=3840

**Stack Frame Size for Each fact() Call**: Assume each call to fact() consumes a stack frame of size S (which includes local variables, return addresses, etc.). Let's estimate that each call to fact() consumes approximately **16 bytes** (reasonable for function call overhead on 32-bit systems).

* **Maximum Recursive Calls**:
  + The maximum number of recursive calls N the program can make before running out of stack space is:

Maximum calls=3840 bytes/16 bytes=240

Therefore, the maximum value of **N** for which the program works correctly is **240**.

**iv. For a value of N greater than this maximum value, what is the likely output and why does it occur?**

* **Answer**: For values of N greater than 240, the program is likely to **crash** due to a **stack overflow**.
  + Since the stack grows downwards, exceeding the allocated 4KB of stack space will overwrite memory outside the stack segment. This usually causes the operating system to raise a **segmentation fault** or **stack overflow error**, crashing the program.
  + The likely output would be a crash, with a message like **"Segmentation fault"** (on Linux/Unix systems), as the program attempts to write to memory outside the allowed stack segment. This occurs because there is no more space left in the stack to allocate frames for additional recursive calls.

**3.**

**(a) W ⊕ X** → **(iii) Prevents execution from certain memory pages**

* **Explanation**: W ⊕ X refers to the **Write XOR Execute** protection (also known as **NX bit** or **DEP (Data Execution Prevention)**). It ensures that memory pages can either be writable or executable, but not both at the same time. This helps prevent the execution of code from areas of memory marked as writable.

**(b) Canaries** → **(i) Enabled mainly by the compiler**

* **Explanation**: **Canaries** are a stack protection mechanism inserted by the compiler to detect buffer overflows. A special value (canary) is placed between the buffer and the control data (return address), and the value is checked before returning from the function. If the canary is altered, the program terminates, indicating a buffer overflow attempt.

**(c) ASLR** → **(ii) Enabled mainly by the Operating System**

* **Explanation**: **Address Space Layout Randomization (ASLR)** is a security feature that randomizes the memory address space of key program areas (e.g., stack, heap, libraries) each time a program is executed. This makes it harder for attackers to predict where their code or return addresses will reside. ASLR is managed by the operating system.

**4.**

To define the users, groups, objects, and access policies for the CS1100 course platform based on the given requirements, we can organize the access control using Linux-style user and group permissions (read, write, execute). Here's a breakdown of the system:

**1. Users:**

* **Teachers**: There are 4 teachers who create and manage quiz materials, view answer scripts, and edit grade sheets.
* **TAs**: There are 20 TAs who assist in reviewing answer scripts but do not have edit access.
* **Students**: There are 400 students who can only view their own answer scripts and view final grade sheets.

**2. Groups:**

To manage access efficiently, we create the following user groups:

* **teachers**: Group for the 4 teachers.
* **TAs**: Group for the 20 TAs.
* **students**: Each student will be a member of the general student group and will also have a unique personal group for individual answer script access.

**3. Objects:**

The objects in the system can be categorized as:

* **Question papers**: Files created by teachers for quizzes, which become visible to all after the exam.
* **Answer scripts**: Files that store individual student answers. Teachers and TAs can view them, but only the specific student can view their own script.
* **Grade sheets**: Files that store the grades. Editable by teachers but viewable by all (students, TAs, teachers).

**4. Access Policies:**

Access permissions are defined using Linux's **read (r)**, **write (w)**, and **execute (x)** permissions.

| **Object** | **Owner** | **Group** | **Others** | **Access Control Policy** |
| --- | --- | --- | --- | --- |
| **Question papers** | Teachers | teachers | Everyone | - Before the exam: rwx for owner and group, --- for others. - After the exam: rwx for owner and group, r-- for others. |
| **Answer scripts** | Individual | teachers, TAs | Students | - Teachers and TAs: r-- (view only). - Individual student: r-- (view only) for their own script. - Others: ---. |
| **Grade sheets** | Teachers | teachers | Everyone | - Teachers: rw- (editable by teachers). - Others: r-- (viewable by all). |

**5. Detailed Policy Implementation:**

1. **Question Papers**:
   * Before the exam: Only teachers can view or edit (rwx) question papers. Students and TAs have no access.
   * After the exam: Question papers are made readable (r--) to everyone.

**Access Control Example**:

bash

Copy code

chmod 770 question\_paper.txt # Before exam (only teachers can read/write/execute)

chmod 774 question\_paper.txt # After exam (everyone can read, only teachers can edit)

1. **Answer Scripts**:
   * Teachers and TAs can view (r--) but not edit answer scripts.
   * Students can only view (r--) their own answer script and not anyone else's.

**Access Control Example**:

bash

Copy code

chmod 640 answer\_script\_student1.txt # Teacher can read, TAs can read, student1 can read their own script

setfacl -m u:student1:r-- answer\_script\_student1.txt # Ensure only student1 can access their script

1. **Grade Sheets**:
   * Only teachers can edit (rw-) the grade sheets.
   * All users (students, TAs, teachers) can view (r--) the grade sheets once published.

**Access Control Example**:

arduino

Copy code

chmod 644 grade\_sheet.txt # Teachers can read/write, everyone else can only read

**6. Summary:**

* **Teachers**: Can create and edit question papers, view all answer scripts, edit and publish grade sheets.
* **TAs**: Can view all answer scripts but not edit them.
* **Students**: Can only view their own answer script and not others'. Can view published grade sheets.

5.

Here’s the correct matching of mechanisms to policies:

* **[A] Hardware Interrupt** → **[ii] Availability of CPU to processes**
  + **Explanation**: Hardware interrupts are used to ensure the **availability** of the CPU to processes by allowing the CPU to respond to hardware events (like I/O operations or timers) that require immediate attention.
* **[B] CPU rings** → **[i] Isolate OS from user processes**
  + **Explanation**: **CPU rings** define different privilege levels in the system, with the OS running in a more privileged ring (Ring 0) and user processes running in less privileged rings (Ring 3). This isolates the OS from user processes, preventing user processes from directly interfering with OS operations.
* **[C] Paging** → **[iv] Isolate user processes**
  + **Explanation**: **Paging** is a memory management technique that isolates user processes by giving each process its own virtual memory space. This prevents one process from accessing the memory of another process, thus ensuring memory isolation.
* **[D] Fat pointers** → **[iii] Memory Buffer Checks**
  + **Explanation**: **Fat pointers** are pointers that carry additional information, such as bounds, which help in performing **memory buffer checks** to prevent buffer overflows and ensure safe memory access.

6.

The primary vulnerability in the given program is a **format string vulnerability** in the printf function:

printf(name);

The printf() function is used without a format specifier (such as %s). If the input for name contains format specifiers (like %x, %n, etc.), they will be interpreted by printf, leading to **unintended behavior**. Specifically, attackers can exploit this to:

* **Read arbitrary memory** using format specifiers like %x.
* **Write to arbitrary memory** using %n, potentially altering variables or control flow.

**Exploiting the Vulnerability:**

An attacker can exploit this format string vulnerability by providing a specially crafted input for the name variable. This input could manipulate the memory in such a way that it modifies the **lottery\_winner** variable or other critical memory locations.

**Steps to Exploit:**

1. **Reading Memory**:
   * An attacker can input format specifiers like %x to read values from the stack or memory.
   * Example input: "%x %x %x %x". This will cause printf() to print memory contents, which could leak sensitive information, such as addresses or variable values.
2. **Writing to Memory**:
   * The format specifier %n allows writing the number of characters printed so far to a specified memory location. This can be used to modify values in memory, including the **lottery\_winner** variable.
3. **Winning the Lottery**:
   * To exploit the program and modify lottery\_winner to match guess, the attacker can craft input using %n to write the desired value to the lottery\_winner variable.

Here's a potential exploit:

* + If the attacker knows the memory address of lottery\_winner, they could input something like this: "%x%x%x%x%n"

Using the %n format specifier, they can overwrite lottery\_winner with the number of characters printed so far, which they can control.

**Example Attack:**

1. **Craft Input**:
   * The attacker provides an input for name such as: "%x%x%x%x%n"
   * The %x specifiers will read data from the stack, and %n will write to a memory location.
2. **Guess the Value**:
   * The attacker can then provide a guess value. If successful, the program will print "You have won the lottery," indicating the lottery\_winner has been altered to match the guess.

**Solution to Fix:**

To prevent this type of vulnerability, the program should always use a format specifier with printf() when printing user-controlled data:

printf("%s", name);

This ensures that the input is treated as a string and does not interpret format specifiers from user input, thereby preventing format string vulnerabilities.

7.

**(a) Give an example of an entity that can be a subject as well as an object.**

* **Example**: A **process** in a computer system.
  + As a **subject**, the process can initiate actions like reading and writing files, executing programs, or sending network requests.
  + As an **object**, it can be acted upon by other subjects, such as when another process queries its state, terminates it, or sends it signals.

**(b) There are several advantages of incorporating access control policies in the hardware. What are the disadvantages?**

* **Advantages** of hardware-based access control:
  + Faster enforcement due to the low-level execution in hardware.
  + Reduced software overhead and reliance on the operating system.
  + Increased difficulty for attackers to bypass security mechanisms.
* **Disadvantages**:
  + **Complexity**: Implementing sophisticated access control in hardware can significantly increase the complexity of the hardware design.
  + **Lack of flexibility**: Hardware is less adaptable compared to software, so updating or modifying access control policies requires redesign or firmware updates, which is costly.
  + **Cost**: Incorporating access control into hardware increases development and manufacturing costs.
  + **Scalability**: It may not scale well with evolving software applications that require different or more complex access control policies.

**(c) Two processes P1 and P2 execute simultaneously in a computer system. What prevents P1 from invoking an arbitrary function in P2?**

* **Answer**: **Memory protection mechanisms** and **process isolation** enforced by the operating system prevent P1 from accessing or invoking functions in P2. These mechanisms include:
  + **Virtual memory**: Each process has its own virtual address space, isolating it from other processes.
  + **Access control**: The OS restricts a process’s access to memory or resources owned by another process.
  + **CPU privilege levels**: User-mode processes cannot directly manipulate the memory or execution of other processes, as that would require kernel-mode access.

**(d) Each user in a Unix system is assigned a user identifier. How is this number assigned?**

* **Answer**: The **user identifier (UID)** in Unix systems is typically assigned by the system administrator or during the creation of the user account via system tools (e.g., useradd). The UID is stored in the /etc/passwd file along with the user's account details. By default, UIDs start from a predefined number (often 1000) for normal users, with lower numbers (0 for root) reserved for system users.

**(e) On a Linux server, user U1 belongs to group G1 and user U2 belongs to group G2. User U1 has a directory /home/U1/team and wants to permit U2 to list the files in the directory but not read the contents of any file in that directory. Write the Linux commands by which U1 can achieve this.**

1. **Grant U2 permission to list files in the directory**:
   * U1 can set the **execute (x)** and **read (r)** permissions for U2 (belonging to group G2) on the /home/U1/team directory.

**Commands**:

setfacl -m u:U2:--x /home/U1/team # Allow U2 to access the directory but not list files yet.

setfacl -m u:U2:r-x /home/U1/team # Grant U2 the ability to list the files in the directory.

1. **Prevent U2 from reading the contents of files**:
   * Ensure the files in /home/U1/team have **no read** (r) permission for U2. The --x permission will allow U2 to navigate the directory without reading file contents.

**Command**:

setfacl -m u:U2:--- /home/U1/team/\* # Remove read permission on all files for U2.

This configuration allows U2 to list files in /home/U1/team but not read their contents.