

**Question 1:** Provide brief answers (within 100 words): [5 X 2 = 10] (a) What are the concerns with penetrate-and-patch approach?

The penetrate-and-patch approach involves identifying vulnerabilities post-deployment and patching them after discovery. It is reactive, not proactive. This method is problematic because:

- Attackers often discover and exploit vulnerabilities before developers patch them.
- Patches may introduce new bugs or security flaws.
- Organizations may delay deploying patches, extending the *window of vulnerability*.
- It fails to address root causes and doesn't promote secure design or coding practices.

(b) What is meant by zero trust? Does it offer attack resistance or attack resilience?

Zero Trust is a security model where no user, device, or system—inside or outside the network—is automatically trusted. It mandates strict identity verification and access control.

**Zero Trust** offers **attack resistance**, as it minimizes the attack surface by continuously authenticating and authorizing every request. However, it does not inherently guarantee resilience (the ability to recover post-attack)

(c) Differentiate temporal and spatial memory safety with help of examples.

- **Temporal Memory Safety:** Prevents use of memory after it's freed.  
**Example:** Using a pointer after free(p) (use-after-free vulnerability).
- **Spatial Memory Safety:** Prevents accessing memory outside the allocated bounds.  
**Example:** Writing past an array boundary (arr[10] = x when arr has only 5 elements).

Both are critical for preventing memory corruption and exploits like buffer overflows.

(d) What are relative strengths of symmetric and asymmetric encryption?

- **Symmetric Encryption:**
  - Fast and efficient for large data.
  - Key management is challenging due to shared secret.
- **Asymmetric Encryption:**
  - Uses public/private key pairs.
  - Enables secure key exchange and digital signatures.
  - Slower but better for secure communication setup.

Symmetric is best for speed; asymmetric is preferred for secure key distribution

(e) Do you think NoSQL databases are more secure than SQL databases?

No. NoSQL databases are **not inherently more secure**. While they may reduce risks like SQL injection, they introduce **new challenges**:

- Often lack mature access control and auditing mechanisms.
- Schemeless design can lead to inconsistent validation.
- Security heavily depends on proper configuration and external controls. Security depends more on **implementation and hardening** than database type.

**Question 2:** Do you think security testing has to be performed manually, or can we automate it?

Security testing should involve **both automated and manual approaches**, each complementing the other:

### **Automated Security Testing**

Automated tools are essential for:

- **Static Analysis (SAST):** Analysing source code for vulnerabilities (e.g., buffer overflows, SQL injections) without running the program.
- **Dynamic Analysis (DAST):** Testing running applications for runtime issues.
- **Tools:** Nmap, Burp Suite, Metasploit, SonarQube, and commercial SAST/DAST tools.
- **Advantages:**

- Fast and repeatable
  - Scalable for large codebases
  - Useful in CI/CD pipelines
- 

## **Manual Security Testing**

Manual testing is required for:

- **Logic flaws** not detectable by tools
  - **Authentication and authorization bypass**
  - **Business logic abuse**
  - **Exploratory testing** using attacker mindset
  - Review of **misuse cases** and **security assumptions**
- 

## **Conclusion**

Automation enhances **efficiency and coverage**, but **manual testing is irreplaceable** for deep, context-aware analysis. A hybrid strategy is essential for robust security assurance.

**Question 3:** List significant aspects of the attack surface of a Web Application.

The **attack surface** of a web application refers to all possible points where an attacker can attempt to exploit vulnerabilities. Key aspects include:

---

### **1. HTTP Interfaces**

- GET, POST, PUT, DELETE endpoints
- Exposed URLs with parameters
- RESTful APIs or GraphQL endpoints

### **2. User Input Fields**

- Form fields (e.g., login, search, registration)
- File upload points
- Query parameters and cookies

### 3. Authentication and Session Management

- Login/logout mechanisms
- Password reset functionality
- Session IDs in URLs or cookies

### 4. Client-Side Scripts

- JavaScript (e.g., XSS vectors)
- AJAX/WebSocket communication
- DOM manipulation

### 5. Third-Party Components

- Plugins, libraries (e.g., jQuery, Bootstrap)
- CDN-hosted scripts
- APIs integrated with external services

### 6. Error Messages and Debug Info

- Verbose error messages revealing stack traces or database queries

### 7. Configuration and Metadata

- Misconfigured HTTP headers (e.g., CORS)
- Exposed .git, .env, or backup files

### 8. Server-Side Resources

- Admin panels
- Database interfaces
- Hidden endpoints (e.g., /debug, /test)



#### **Best Practice:**

Reduce the attack surface by disabling unused features, enforcing input validation, and applying **principle of least privilege**

**Question 04:** Briefly explain process memory organization for Unix or Windows machine. Identify parts of process memory that can potentially be exploited by an attacker. Justify your answer.

A process's memory in Unix or Windows is typically organized into the following segments:

---

## Memory Segments

Segment	Description
Text (Code)	Contains compiled code (instructions) of the program. Usually read-only and executable.
Data	Stores global and static variables initialized by the program.
BSS	Holds uninitialized global and static variables.
Heap	Used for dynamic memory allocation (e.g., malloc in C, new in C++). Grows upward.
Stack	Stores function call frames, local variables, return addresses. Grows downward.
Memory-mapped	Used for shared libraries, files, etc.

---

## Exploitable Areas

### 1. Stack

- **Vulnerable to:** Buffer overflows, stack smashing.
- **Why:** If input exceeds allocated space, it can overwrite return addresses or control data (e.g., injecting shellcode).

### 2. Heap

- **Vulnerable to:** Heap overflows, use-after-free, double free.
- **Why:** Attackers can overwrite dynamic memory structures (e.g., malloc metadata) to hijack control flow or escalate privileges.

### 3. Memory-mapped Region

- **Vulnerable to:** Code injection via shared library misuse or DLL hijacking.
-

## Justification

Attackers exploit **stack and heap** because they are writable and handle user-controlled input. If unchecked, these regions can be manipulated to:

- Redirect program execution
- Inject malicious code
- Access unauthorized memory

Hence, secure coding practices and protections like **stack canaries**, **DEP**, and **ASLR** are critical defences.

**Question 05:** Compare security concerns in Python with that of C language?


**Answer: Q5 – Comparison of Security Concerns in Python vs C Language**

Aspect	Python	C Language
Memory Management	Automatic (Garbage Collected)	Manual (e.g., malloc, free) – prone to leaks, dangling pointers
Buffer Overflows	Highly unlikely (no raw pointers or manual indexing)	Common vulnerability due to unbounded array operations
Type Safety	Strong dynamic typing; type errors raise exceptions	Weakly typed; allows type casting and pointer arithmetic
String Handling	Safe string manipulation (str objects)	Insecure functions like strcpy, gets cause buffer overruns
Pointer Usage	No raw pointers exposed to user	Direct memory access – risk of unauthorized memory manipulation
Interpreter-level Security	Can sandbox or restrict execution (e.g., exec, eval risks)	No interpreter; relies entirely on secure coding practices

Aspect	Python	C Language
Use in Scripting/Automation	Often used for scripting, may embed sensitive logic	Less common for scripting; usually used in compiled programs

### Summary:

- **C** is vulnerable to **low-level attacks** (e.g., buffer overflow, memory corruption) due to manual memory control and unsafe APIs.
- **Python** abstracts these risks but has **runtime-level concerns** like misuse of `eval()` or insecure deserialization (`pickle`).

 **Conclusion:** Python is generally safer by design, but not immune. C offers performance but requires much stricter discipline for secure coding.

**Question 06:** List concerns, if any, with the program given below. Justify. [5 Marks]

```
#include <stdio.h>
#include <string.h>

void welcome_func(char *name)
{
    char buf[30];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    welcome_func(argv[1]);
    return 0;
}
```

## Security Concerns

### 1. Buffer Overflow (Critical)

- `char buf[30];` creates a fixed-size buffer of 30 bytes.
  - `strcpy(buf, name);` does **not check the length** of name.
  - If `argv[1]` (input) exceeds 29 characters (plus null terminator), it **overwrites adjacent memory**.
  - This classic **stack-based buffer overflow** can lead to:
    - Program crash
    - Arbitrary code execution
    - Potential privilege escalation
- 

## 2. Lack of Input Validation

- No check is done to ensure `argv[1]` is not NULL.
  - If no command-line argument is passed, `argv[1]` will be NULL, causing segmentation fault.
- 

## 3. Format String Vulnerability (Possible Extension)

While not exploited here, using:

```
printf(buf); // ← would be vulnerable if written like this
```

Instead of:

```
printf("Welcome %s\n", buf); // ← safe
```

...would allow an attacker to inject format specifiers like `%x`, `%n`.

## Mitigation Recommendations

- Use `strncpy()` or `snprintf()` to avoid overflows:

```
strncpy(buf, name, sizeof(buf) - 1);
```

```
buf[sizeof(buf) - 1] = '\0';
```

- Validate that `argc > 1` before accessing `argv[1]`.
  - Avoid unsafe functions like `strcpy()` in security-sensitive applications.
- 

## Conclusion



The program is **vulnerable to buffer overflow**, a serious security flaw. Without bounds checking, it is exploitable and must be rewritten using safe functions and proper input validation.

**Question 07:** An ecommerce site stores product details in an RDBMS. It allows user to enter product group, captures into a variable named prodGroup and lists name and manufacturer (both are of type varchar) for the available products in the product group. They generate the SQL statement by appending prodGroup to the string "SELECT name, manufacturer FROM Products WHERE group =". Further prodGroup is validated for absence of special characters.

(a) Do you notice any flaws/weaknesses in the software design? What can we do about it?

### Given Scenario:

- The application uses the following pattern:

```
"SELECT name, manufacturer FROM Products WHERE group = " +  
prodGroup
```

- prodGroup is validated only for absence of special characters
- name and manufacturer are VARCHAR

### (a) Design Flaws and What Can Be Done

#### Flaws in Design:

##### 1. String Concatenation to Build SQL Queries:

- Even though special characters are filtered, this practice is unsafe.
- Filtering can be bypassed or incomplete (e.g., Unicode tricks, encoding evasion).
- It's still susceptible to **SQL Injection** via clever inputs.

##### 2. Validation ≠ Sanitization:

- Absence of special characters doesn't ensure the input is free from malicious intent (e.g., logic injection like OR 1=1).

#### Mitigation:

- **Use Prepared Statements / Parameterized Queries:**

```
cursor.execute("SELECT name, manufacturer FROM Products WHERE  
group = ?", (prodGroup,))
```

- **Implement Role-Based Access Control (RBAC)** to prevent unauthorized table access.
- **Whitelist allowed values** of prodGroup from a trusted set (e.g., ['Electronics', 'Books']).

(b) Is it possible for an attacker to obtain usernames and passwords stored in another table Users by crafting input for the product group? [5 Marks]

✓ **Yes, It's Possible – Here's How:**

If prodGroup is directly injected and not fully sanitized, an attacker could input:

```
'Electronics' UNION SELECT username, password FROM Users--
```

This would yield:

```
SELECT name, manufacturer FROM Products WHERE group = 'Electronics'  
UNION SELECT username, password FROM Users--'
```

**Effect:**

- Returns name and manufacturer from Products
- Followed by username and password from Users (column count matches!)
- -- comments out any trailing syntax error

🔵 **Conclusion:**

- The current design is vulnerable to **SQL Injection**, even with special character validation.
- The attacker can indeed extract sensitive data like usernames/passwords from other tables unless **prepared statements and proper access controls** are enforced.

**Question 08:** You are the CISO (Chief Information Security Officer) of an organization. List the events (in software environment) that you would like to be alerted promptly. Provide a brief justification for each event. [5 Marks]

As a **CISO**, timely alerts are vital to detect, respond, and contain security incidents before they escalate. Below are **key software-related events** that should trigger alerts, along with justifications:

### 1. Unauthorized Access Attempts (Login Failures, Privilege Escalation)

- **Why?** May indicate brute-force attacks, compromised credentials, or insider threats.
- **Example:** Multiple failed logins or a regular user suddenly trying to access admin functions.

### 2. Modification of Critical Configuration or Source Code

- **Why?** Indicates possible tampering or insider attacks aiming to insert backdoors or malware.
- **Example:** Unexpected Git commits or changes to security policy files (firewall.conf, .env).

### 3. Execution of Suspicious or Unusual Software

- **Why?** Could indicate malware or ransomware being triggered within the system.
- **Example:** Scripts running from temp folders or tools like nmap, netcat being executed on production servers.

### 4. Unexpected Access to Sensitive Data (PII, financial, credentials)

- **Why?** Could be data exfiltration or reconnaissance activity by a malicious actor.
- **Example:** Large-scale queries on Users, Payments, or Secrets tables outside business hours.

### 5. Security Control Failures (Firewall Disabled, Logging Turned Off)

- **Why?** Attackers often disable defences to cover their tracks; early alert enables immediate remediation.
- **Example:** A process stopping the intrusion detection system (IDS) or altering logging configurations.

#### Summary:

These alerts help detect **breach attempts, data misuse, and insider threats**. Early detection is key to minimizing **damage, downtime, and legal/regulatory consequences**.