PRACTICAL - 1

AIM: Identify common threats to software security, study sources of software insecurity and potential risks.

Solution:

Introduction

Software security is the practice of protecting software against unauthorized access, use, disclosure, disruption, modification or destruction. It includes understanding the **threat landscape**, identifying internal and external sources of insecurity and implementing preventive measures.

Software Security Threats

1. Insider Threats

Insider threats originate from individuals within the organization or system, such as employees, contractors or partners who have authorized access to critical systems or data. These threats result from intentional abuse of privileges or accidental actions that compromise security.

Examples:

- **Developer leaking source code:** A programmer intentionally or inadvertently uploads proprietary source code to a public repository or shares it with unauthorized personnel.
- **Administrator misusing privileged access:** A system admin uses root or elevated privileges to access confidential data they are not authorized to view or modifies logs to hide suspicious activity.
- **Insider altering system configurations:** An employee changes firewall rules, disables security controls or alters application settings, intentionally or by mistake, leading to vulnerabilities.

2. External Threats

External threats are posed by actors outside the organization such as hackers, cybercriminals, or state-sponsored attackers, who attempt to compromise the system without authorized access.

Examples:

- **Hackers exploiting open ports:** Attackers scan for and target servers with open or weakly protected ports, attempting to breach the system using vulnerabilities.
- **Malware delivered through phishing emails:** Cybercriminals send deceptive emails with infected attachments or malicious links, tricking users into installing software like ransomware or spyware.
- **Remote attackers injecting malicious code:** External adversaries leverage web application flaws (e.g., SQL injection, Cross-Site Scripting) to execute unauthorized code, access restricted data or compromise user accounts.

Common Threats to Software Security

Threat	Description	Example Code / Scenario	Mitigation Strategy	Type
Malware	Malicious software (virus, worm, trojan, ransomware) infects the system, stealing or corrupting data.	Users download a trojan disguised as legitimate software	Use antivirus tools, educate users, regularly update and patch systems	**External**
Spyware	Software that secretly monitors or collects users' data.	Unwanted browser toolbars or background apps collecting keystrokes	Use anti-spyware software, review app permissions, restrict downloads	**External**
SQL Injection	Attacker injects malicious SQL to bypass login or extract DB data.	SELECT * FROM users WHERE username='$user' AND pass='$pass'; Input: ' OR 1=1; --	Use prepared statements/parameterized queries, validate input	**External**
Cross-Site Scripting	Injected JavaScript runs in the victim's browser.	<input value="<script>alert(1)</script>"/>	Encode user output, use Content Security Policy (CSP)	**External**
Buffer Overflow	Exploits unbounded memory storage to crash or inject code.	In C: char buf; gets(buf);	Bounds checking, use safe libraries (e.g., fgets), avoid unsafe functions	**External**
Hardcoded Credentials	Developer includes passwords in code which can be leaked.	String password = "admin123";	Use environment variables or secure vaults for secrets	**Internal**

Unvalidated Input	Failing to check inputs leads to injections, overflows, etc.	No validation on form fields; user submits malicious data	Validate and sanitize all input data, enforce data types	**Both**
Insecure File Uploads	Malicious files uploaded that get executed on the server.	User uploads webshell.php as profile pic	Check file type/size, store outside webroot, scan uploads	**External**

Sources of Software Insecurity

- **Poor Coding Practices:** Lack of input validation, hardcoded credentials, improper memory management.
- **Inadequate Testing:** Not performing proper security or vulnerability testing, focusing only on functionality.
- **Misconfigurations:** Using insecure default settings, open ports, or unnecessary enabled services.
- **Unpatched Vulnerabilities:** Not applying updates or security patches.
- **Third-Party Components:** Relying on vulnerable libraries/frameworks.
- **Human Factors:** Social engineering, insider threats and inadequate training.

Case Study Analysis: Equifax Data Breach

Background: In 2017, Equifax (a major US credit reporting agency) suffered one of the largest data breaches in history, affecting 147 million people. Attackers exploited a vulnerability in the Apache Struts web framework used by Equifax.

- **Vulnerability:** Failure to patch the Apache Struts CVE-2017-5638 vulnerability.
- **Threat:** External (attackers targeting publicly accessible web applications).
- **Attack Vector:** Attackers sent malicious requests to Equifax's servers, exploiting a remote code execution flaw.
- **Impact:**
    - Leak of sensitive personal and financial information including Social Security Numbers (SSNs), addresses and driver's license details.
    - Massive financial loss ($575M in settlements and fines).
    - Severe reputational damage.
- **Lessons Learned:**
    - Always apply security patches as soon as they are released.
    - Regular vulnerability assessments and penetration testing are critical.
    - Use monitoring and intrusion detection systems (IDS) to detect suspicious activities.
- **Vulnerability Testing:**
    - Static Analysis: Review source code for vulnerabilities.
    - Dynamic Analysis: Analyze running applications for security issues.
    - Penetration Testing: Simulate attacks to identify real-world risks.
    - Continuous Monitoring: Use automated tools to maintain ongoing security.

Potential Risks and Impacts

Risk	Impact
Data Breach	Leakage of sensitive information (PII, passwords, etc.)
Financial Loss	Ransom payments, lawsuits or regulatory fines
Reputational Damage	Loss of customer trust
Intellectual Property Theft	Competitors gaining access to proprietary systems
Service Disruption	Downtime affecting business operations
Legal Consequences	Violations of data protection laws (e.g., GDPR, HIPAA)

Risk Assessment Matrix

Threat	Likelihood	Impact	Risk Level	Mitigation Strategy
Malware Infection	Medium	High	High	Antivirus, patch management
Spyware	Medium	Medium	Medium	Educate users, endpoint protection
SQL Injection	High	High	Critical	Use parameterized queries/input validation
XSS (Cross-site Scripting)	Medium	Medium	High	Encode outputs, use CSP
Hardcoded credentials	Low	High	High	Policy enforcement, code review
Unpatched Libraries	High	High	Critical	Dependency management, regular updates
Insecure File Upload	Medium	High	High	File validation, storage outside webroot

Sample Codes – Demonstrating a Vulnerability

Example 1: SQL Injection in PHP (Vulnerable Code)

```php
<?php

// vulnerable.php

$username = $_GET['user'];

$password = $_GET['pass'];



$conn = mysqli_connect("localhost", "root", "", "test_db");

$query = "SELECT * FROM users WHERE username='$username' AND password='$password'";

$result = mysqli_query($conn, $query);



if (mysqli_num_rows($result) > 0) {

   echo "Login Successful";

} else {

   echo "Invalid Login";

}

?>
```

Problem: If attacker enters - user=admin' – &pass=abc
It becomes: SELECT * FROM users WHERE username='admin' -- ' AND password='abc';
The `--` comments out the rest, giving unauthorized access.

Solution (Using Prepared Statements):

```php
<?php

// secure.php

$username = $_GET['user'];

$password = $_GET['pass'];

$conn = mysqli_connect("localhost", "root", "", "test_db");

$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");

$stmt->bind_param("ss", $username, $password);

$stmt->execute();

$result = $stmt->get_result();
```

```
if ($result->num_rows > 0) {
   echo "Login Successful";
} else {
   echo "Invalid Login";
}
?>
```

Example 2: Cross-Sites Scripting in Javascript/HTML (Vulnerable Code)

```
<!-- User input rendered directly into page without sanitization -->
<form>
   <input name="username" value="<?php echo $_GET['username']; ?>">
</form>
```

Problem: If the URL is ?username=<script>alert('XSS')</script>, the script executes in the victim's browser window.

Solution: Using htmlspecialchars escapes special characters, preventing scripts from executing.

```
<!-- Sanitizes user input before rendering in HTML -->
<form>
     <input name="username" value="<?php echo htmlspecialchars($_GET['username'],
ENT_QUOTES, 'UTF-8'); ?>">
</form>
```

Example 3: Buffer Overflow in C (Vulnerable Code)

```
#include <stdio.h>
int main()
{
   char buf[10];
   gets(buf); // Unsafe: gets() allows unlimited input
   printf("%s\n", buf);
   return 0;
}
```

Problem: Input longer than 10 characters can overflow the buffer and hijack program execution.

Solution: fgets enforces buffer boundary checks, preventing overflows.

```c
#include <stdio.h>

int main()

{

   char buf[10];

   fgets(buf, sizeof(buf), stdin); // Safe: limits input size

   printf("%s\n", buf);

   return 0;

}
```

Example 4: Hardcoded Credentials (Vulnerable Code)

```python
# insecure_login.py

USERNAME = "admin"

PASSWORD = "admin123"

input_user = input("Username: ")

input_pass = input("Password: ")

if input_user == USERNAME and input_pass == PASSWORD:

   print("Login successful!")

else:

   print("Access Denied")
```

Problem: Anyone with code access can see the credentials.

Solution: Use .env files or secure credential storage:

```
# .env

USERNAME=admin

PASSWORD=admin123
```

```python
# secure_login.py

import os

from dotenv import load_dotenv

load_dotenv()

username = os.getenv("USERNAME")

password = os.getenv("PASSWORD")

input_user = input("Username: ")
```

```
input_pass = input("Password: ")

if input_user == username and input_pass == password:

   print("Login successful!")

else:

   print("Access Denied")
```
