Infomation

Question	Question title	Marks	Question type
ℹ	Welcome		Information or resources

Quick questions

Question	Question title	Marks	Question type
1	Quick questions	12	Text area

Cross-site Scripting

Question	Question title	Marks	Question type
2	Cross-site scripting (XSS)	11	Text area

Security of password authentication

Question	Question title	Marks	Question type
3	Authentication, requirements and assumptions and design	15	Text area

Access control

Question	Question title	Marks	Question type
4	Access control	9	Text area

A webmail program

Question	Question title	Marks	Question type
5	Webmail	13	Text area

Closing thoughts

Question	Question title	Marks	Question type
6	Not a question	0	Text area

Welcome

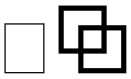
Welcome to the exam of INF226.

This exam has a total of five tasks, each divided into several questions.

This is an on campus digital exam, without any permitted aids.

- This exam counts for 60% of your final grade in this course.
- Points on the exercises indicate *approximate* percentage weight on the total final grade.
- Give justifications for your answers, unless otherwise specified.
- Use precise language and make sure to read the exercises carefully.

Good luck!



1 Quick questions

Answer the following six questions in one or two sentences each.

a) What is the primary purpose of a stack canary in a program's memory?

b) Why should untrusted input never be concatenated into a SQL query?

c) What kind of vulnerabilities do the SameSite=Strict attribute for cookies help prevent?

d) Why do we add salt to a password when hashing it for storage?

e) Which techniques do we have to prevent cross-site request forgery (CSRF)?

f) Why should untrusted input never be concatenated into HTML?

Maximum marks: 12

2 Cross-site scripting (XSS)

```
CODE TABLE 0
<script>
  // Rich text editor code
  const editor = document.getElementById("commentEditor");
  const submitButton = document.getElementById("submitComment");
  submitButton.addEventListener("click", () => {
    const userInput = editor.htmlContent; // User's input as html
    // Cancel submission if <script> is detected
    if (userInput.includes("<script")) {
      alert("Your comment contains prohibited content!");
      return;
    }
    // Send HTML to the backend
    fetch("/submitComment", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ comment: userInput }),
    });
  });
</script>
```

This task has 4 questions, each to be answered in individual boxes.

You are working as a developer for a blogging platform called **InWrite**, which allows users to write and share blog posts. Each blog post includes a comment section where readers can leave feedback. To improve the user experience, your team implemented a rich text editor in JavaScript for the comment section, enabling users to format their comments (e.g., bold, italic, links).

In CODE TABLE 0 is the code which sends the formatted comment to the back-end.

1. Suggest two possible threats a XSS scripting vulnerability could cause to the security of the blogging platform.

2. Explain why the test for "`<script>`" tags implemented above is insufficient to detect possible script injection in the user input. Give concrete examples of how malicious code can bypass this test.

3. Explain why the client side JavaScript is not a secure place to put the sanitation check.

4.Outline a strategy for securing the InWrite blogging platform against XSS vulnerabilities. Take into account the challenges caused by the comment text being represented by HTML.

Maximum marks: 11

3 Authentication, requirements and assumptions and design

This task has 7 questions, each of which is to be answered in a separate box.

The definition of software security we have worked with in this course is: **"Software security is the ability of software to function according to intentions in an adversarial environment"**. In order to judge software to be secure one must make assumptions on the adversarial environment, create requirements which codify the intended function of the program, and design mechanisms which ensure the requirements are kept given the assumptions made.

In this task we will investigate how this plays out for authentication. We will look specifically at password authentication. Let us take the following as the fundamental requirement of authentication:

Fundamental requirement of authentication: *"Only a human with legitimate access to an account will be able to log in."*

For password authentication, the mechanisms we use also imply that *"being able to log in to an account" is a consequence of "knowing the password of that account"*, so we will take this as a fundamental assumption. Thus, for the fundamental requirement to hold, we must ensure that only humans with legitimate access know or can guess the password.

Let us start by looking at some assumptions we might make, helping us design our login mechanisms:

- The human who created the account has legitimate access.
- The human who created the account knows the password.
- A human who has legitimate access to an account will only tell the password to another human with legitimate access to that account.

1. Explain how the third assumption in the above list may fail in a way which causes the fundamental requirement to fail.

2. Explain two different situations where the above assumptions hold, but the fundamental requirement still fails. Make sure that your examples are very concrete.

Here are some assumptions which point to potential security issues:

- A human will create a weak password if allowed to.
- A weak password can be guessed by an attacker.

3. Explain carefully how these assumptions could lead to failure of the fundamental requirement.

One of the mechanisms we use in password authentication is rules on the passwords on an account. Common rules concern password length, allowed characters and disallowing certain weak passwords from a dictionary.

4. Give a robust set of password rules and explain how they prevent the failure you described in 3. If needed, you can introduce reasonable additional assumptions.

Assume that our password requirements ensure that all accounts have *reasonably strong passwords*. But even strong passwords can be guessed given enough attempts. There is a limit to how strong passwords humans will go around remembering.

5. What technical mechanisms can we employ in our login to limit the attacker's ability to guess passwords?

Finally, we will look at storage of passwords in the database. A pessimistic, but realistic assumption is:

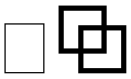
- The database underlying our password authentication can be leaked to an attacker.

Obviously, if we store the passwords for the accounts as plain text in the database, this immediately causes failure of the fundamental requirement of authentication. So, instead, we use a mechanism called a *key derivation function*, such as *Argon2* or *SCrypt*. In addition to being salted, these allow adjusting the computational costs of computing the derived key, by changing the *key derivation function's parameters*.

6. Why is it important that we can adjust both the memory and CPU costs of a key derivation function?

7. Under what assumptions are the accounts protected by using a key derivation function like Argon2 or SCrypt, in the event of a password database leak? Try to take into account the cost vs benefit for an attacker.

Maximum marks: 15



4 Access control

This task has 3 questions, each to be answered in individual boxes.

A university has hired you as a consultant to help improve their access control system. One of the issues Maria, the campus facilities manager, brings up is that their current system is too tedious to manage. At the university, every student and staff member has an access card which is used when entering campus buildings, lecture auditoriums, study halls, offices, and certain restricted areas. The access permissions vary depending on whether the person is a student, master student, TA, lecturer, administrative staff...

The rules for the access cards use an access control list (ACL), which Maria updates manually. Here is the database schema for the ACL:

User(id TEXT)
…

Permission(permission TEXT)
…

UserPermission(user TEXT,	**permission TEXT)**
…	…

UserPermission is subject to the following constraints:

FOREIGN KEY user REFERENCES User(id) FOREIGN KEY permission REFERENCES Permission(permission)

Maria's complaint is that it is very tedious to update the list with individual permissions whenever someone joins, graduates, or changes their role – not to mention the effort required when a new building or facility is added that only certain groups need access to. When you hear the facilities manager's complaint, you immediately think of role-based access control (RBAC) and suggest that it may simplify her task of updating the list.

1. Propose a database schema for RBAC at the university.

2. Once the system has been ported from ACL to RBAC, how would you test that the translation was correct? I.e. that at the end of the transition every user has the same access as they started with. You can write out an SQL query, or just explain the tests you would like to perform concretely with words.

3. Explain the advantages of RBAC over ACL in this case, with respect to security.

Maximum marks: 9

5 Webmail

```
CODE TABLE 1
@GetMapping("/message/{id}")
public ResponseEntity<Map<String, Object>> getMessageById(@PathVariable("id") int id) {
    String sql = "SELECT id, sender, recipient, subject, body FROM messages WHERE id = ?";
    try (Connection connection = dataSource.getConnection();
         PreparedStatement statement = connection.prepareStatement(sql)) {
        statement.setInt(1, id);
        try (ResultSet resultSet = statement.executeQuery()) {
            if (resultSet.next()) {
                Map<String, Object> message = new HashMap<>();
                message.put("id"       , resultSet.getInt("id"));
                message.put("sender"   , resultSet.getString("sender"));
                message.put("recipient", resultSet.getString("recipient"));
                message.put("subject"  , resultSet.getString("subject"));
                message.put("body"     , resultSet.getString("body"));
                return ResponseEntity.ok(message);
            } else {
                return ResponseEntity.status(HttpStatus.NOT_FOUND)
                                     .body(Map.of(  "error"
                                                  , "Message not found"));
            }
        }
    } catch (SQLException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body(Map.of( "error"
                                         , "Database error"
                                         , "details"
                                         , e.getMessage()));
    }
}
```

This task has 6 questions, each to be answered in individual boxes.

In this exercise we will imagine a webmail platform, where users can log in to read and send e-mail. E-mails are identified on the server by an integer ID number. The ID numbers are assigned automatically in the data base, using the SQL AUTO_INCREMENT directive. This means that the messages are assigned a global ID, where the first message on the server has ID 0, the next one has ID 1 etc... Message IDs are not grouped by who owns them, so 2043 and 3581 may belong to Alice while 999 and 2044 may belong to Bob. Also, note that the application shares a single connection to the database.

In CODE TABLE 1 you can see the the server side function which (upon request from the user interface (UI)) fetches a message from the database and returns it as JSON to the UI so that it can be displayed to the user.

1. What is the immediate security issue with the above code?

2. How would you classify this vulnerability? What threat does it pose?

Someone on the development team suggests that a good way to fix this security vulnerability would be to use UUIDs instead of numerical IDs. The idea being that an attacker will not know the UUID of other peoples messages.

3. UUIDs come in several version. In each version the total length is 128 bits. Below is a table which summarises the different versions. Which version would you choose for this application, and why?

UUID Version	Description	Components
UUIDv1	Time-based UUIDs using the system clock and node identifier (usually the MAC address).	48-bit timestamp, 16-bit clock sequence, 48-bit node (e.g., MAC address).

UUID Version	Description	Components
UUIDv2	DCE Security version with embedded POSIX UID or GID information.	32-bit timestamp, 14-bit clock sequence, 48-bit node with 8 bits replaced by the POSIX UID/GID info.
UUIDv3	Name-based UUIDs generated using MD5 hashing.	MD5 hash of a namespace identifier and a name, with fixed version and variant bits.
UUIDv4	Randomly generated UUIDs.	Entirely random except for version (4) and variant bits.
UUIDv5	Name-based UUIDs generated using SHA-1 hashing.	SHA-1 hash of a namespace identifier and a name, with fixed version and variant bits.
UUIDv6	Time-ordered UUIDs with reorganized fields for lexicographical sorting.	60-bit timestamp, 48-bit node (MAC address), 14-bit clock sequence.
UUIDv7	Timestamp-based UUIDs with random components, optimized for sorting.	48-bit big-endian Unix Epoch timestamp, version nibble (7), variant bits (10x), remaining 74 bits random.
UUIDv8	Customizable format allowing arbitrary data, conforming to the UUID structure.	Depends on implementation, with 122 customizable bits after version and variant bits.

4. Taking into account the properties of the UUID version you chose, how would changing the IDs to UUIDs impact the security of the above code?

CODE TABLE 2

```
@GetMapping("/message/{id}")
public ResponseEntity<Map<String, Object>> getMessageById(@PathVariable("id") int id, @RequestParam("username")
    String sql = "SELECT id, sender, recipient, subject, body FROM messages " +
                 "WHERE id = " + id + " AND owner = '" + username + "'";
    try (Connection connection = dataSource.getConnection();
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(sql)) {
        if (resultSet.next()) {
            Map<String, Object> message = new HashMap<>();
            message.put("id", resultSet.getInt("id"));
            message.put("sender", resultSet.getString("sender"));
            message.put("recipient", resultSet.getString("recipient"));
            message.put("subject", resultSet.getString("subject"));
            message.put("body", resultSet.getString("body"));
            return ResponseEntity.ok(message);
        } else {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(Map.of("error", "Message not found"));
        }
    } catch (SQLException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body(Map.of("error", "Database error", "details", e.getMessage()));
    }
}
```

A junior programmer on the team overhears you talking about the bug, and implements their own fix. (See CODE TABLE 2)

5. Evaluate the security of the junior programmer's fix.

6. How would you fix the issues in the original code? You do not have to provide the exact code, but explain your solution as detailed and concretely as you can.

Maximum marks: 13

6 Not a question

This is not an exam question, just a note for your entertainment if you are done early.

While writing the XSS task on this exam, whenever I wrote <script> in a question description, the rest of the text would disappear. Naturally, I suspected XSS vulnerabilities in inspera (not the first time!), and a quick test proved this to be correct. In the end I had to manually escape <script> in my text to avoid accidental script injection.

Unless Inspera has fixed their vulnerability in time for your exam, there should be a JavaScript animated heart I have injected below.

Merry Christmas!

Maximum marks: 0