Practical information about the exam

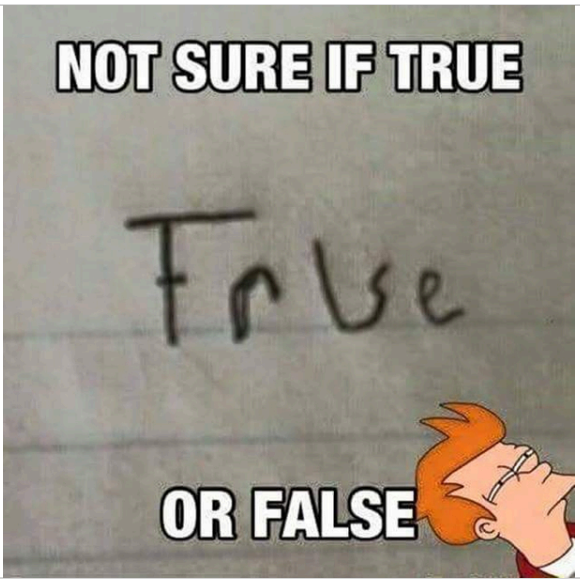
INF226 Exam – Spring 2024

This exam has **6** questions and counts for 60% of your final grade. All parts (a,b,c…) of each exercise should be answered.

- *Allowed support materials:* **any written or printed material** ("open book"). For your convenience, some resources are attached to the exam as PDFs.
- Please *justify* your answers, unless otherwise specified.
- Use precise language and read the exercise text carefully.
- Read through the whole exercise set first so you get a good overview and can plan your time appropriately. Make a note of any questions you might have.
- The percentages indicate the *approximate* weight (out of 60%). The final two questions should **not** be answered: Question 7 is just a placeholder for the compulsory assignment results (40% of the grade), and question 8 is just a demo of the problem from question 5.
- If you get stuck somewhere, take a break or move on to the next exercise – you can always come back later if you have time!
- If you don't know the answer to some part of an exercise, it's ok to assume you've answered it when answering the other parts.
- You might write longer answers than suggested if that's easier for you – but avoid writing several pages instead of several sentences. It's ok to keep the answers fairly brief.

Good luck!

1 1. True or False? [10%]



Select *True* or *False* for each statement, as appropriate. 0.*50 pts for correct answers, –0.25 pts for incorrect answers and no points for unanswered. (Minimum 0 pts total, max 10 pts.)*

1. In a *memory safe* programming language, buffer overflows cannot happen.
Select one alternative:

- True
- False

2. Python is a memory safe language.
Select an alternative

- True
- False

3. C is a memory safe language.
Select an alternative

- True
- False

4. With HTTPS, the client (web browser) can authenticate the server.
Select an alternative

- True
- False

5. HTTPS makes code injection attacks almost impossible in practice.

Select an alternative

- True
- False

6. A good implementation of password authentication should use *salt*.
Select an alternative

- True
- False

7. The *Same-Origin Policy* protects users against third-party cookies.
Select an alternative

- True
- False

8. ASLR (address-space layout randomisation) makes buffer overflows impossible to exploit.
Select an alternative

- True
- False

9. A *same-origin URL* has the same protocol, port and host as the current page.
Select an alternative

- True
- False

10. *Cross-site scripting* attacks are only a problem if the site uses JavaScript.
Select an alternative

- True
- False

11. *Prepared statements* are used to prevent cross-site request forgeries.
Select an alternative

- True
- False

12. “Same-site” is just an older term for “same-origin”.

Select an alternative

- True
- False

13. SQL injection vulnerabilities can often be detected with a static analysis tool.
Select an alternative

- True
- False

14. A *Cross-Site Request Forgery* attack can be made using just HTML code. (Assuming the site is vulnerable.)
Select an alternative

- True
- False

15. Setting a cookie's *SameSite* attribute to *Strict* will prevent cookies from `https://uib.no` from being sent to `https://mitt.uib.no`.
Select an alternative

- True
- False

16. Cookies from `https://mitt.uib.no` will also be sent to `https://uib.no` unless we set the *SameSite* attribute.
Select an alternative

- True
- False

17. SQL queries should always be kept as *short and simple* as possible to prevent SQL injection attacks.
Select an alternative

- True
- False

18. *Application performance* is irrelevant from a security perspective.
Select an alternative

- True
- False

19. Writing code that is *easy to understand* can help prevent security problems.
Select an alternative

- True
- False

20. In case of a security breach where personal data is exposed, European law generally requires that the proper authorities (the *Data Protection Authority* (DPA) – Datatilsynet in Norway) should be informed with 72 hours.
Select an alternative

- True
- False

Maximum marks: 10

2 2. Fill in! [5%]

Fill in the missing text, so that the statements make sense. *1 point for each correct answer, 0 points for missing or wrong answer.*

- Using the [ ] HTTP header, we can prevent injected code from running in the browser.
- Set SameSite=[ ] to prevent cookies from being sent when a user navigates to your site from another site.
- A [ ] is a secret value placed on the stack to help protect against stack smashing / buffer overflow attacks.
- With the help of [ ] , we can let users authenticate to our web site using an external service (e.g., with GitHub, Google, FEIDE, etc.)
- To help protect cookies (e.g., a session cookie) against code injection attacks, we can set the [ ] flag on the cookie. This will make the cookie value inaccessible to client-side JavaScript code (in compliant browsers).

Maximum marks: 5

3 3. SQL Trouble [a/b, 15%]

You and another INF226 student have been hired to work on one of the University's administrative systems. The system has an SQL database with student data, and the backend code is written in Java.
The query code looks something like this:

```
  /** Return a formatted list of the courses the current user is signed up for. */
  public String viewCourses(String filter) throws SQLException {
    String result = "";
    String query = "SELECT * FROM Enrollments WHERE student_id = '"
                 + currentUser.studentId + "'";
    if(filter != null) { // add optional course name filter
        query += " AND course_name LIKE '" + filter + "'";
    }
    query += ";"
    ResultSet rs = connection.execute(query);
    while (rs.next()) {
        result += formatCourse(rs.getInteger("year"),
                        rs.getString("course_name"),
                        rs.getString("grade"));
    }
    return result;
  }
```

a) [7%]
Why is the code (probably) unsafe? **Explain what the vulnerability is, and how you would fix it.**

(2–3 paragraphs)

b) [8%]
Your co-worker suggests that instead of using plain SQL statements, you can use a new Java framework that lets you define database tables and columns in Java and write queries using normal Java method call syntax.
For example, the *enrollment* table might be defined like this:

```
public class Enrollment extends Table {
    // Id values should correspond to student and course tables
    public static Column<Integer> studentId = new IntegerColumn().foreignKey(Student.id);
    public static Column<Integer> courseId = new IntegerColumn().foreignKey(Course.id);
    // The course name is required
    public static Column<String> courseName = new TextColumn().notNull();
    // Grade is optional, but should be a single letter A-F if present
    public static Column<String> grade = new TextColumn().orNull().length(1).matches("[A-F]");
}
```

Each `Column` object contains the definition of the corresponding column in the SQL table. (`Column` and associated interfaces are sketched in the attached PDF.)

Utility methods like `equalTo()`, `like()`, `before()`, `lessThan()`, `orderBy()` etc. make it easy to build SQL queries in a safe manner. For example:

```
/** Return a formatted list of the courses the current user is signed up for. */
public String viewCourses(String filter) throws SQLException {
    String result = "";
    WhereClause query = Enrollment.studentId.equalTo(currentUser.studentId);
    if(filter != null) // we can combine where clauses with 'and()':
        query = query.and(Enrollment.courseName.like(filter));
    // code is sql-like but uses fluent-style Java method calls
    ResultSet = connection.select().from(Enrollment.class).where(query).execute();
    while (rs.next()) {
        // column knows its type, so we don't need to use getString, getInteger, etc
        result += formatCourse(Enrollment.year.get(rs),
                    Enrollment.courseName.get(rs),
                    Enrollment.grade.get(rs));
    }
```

```
    return result;
}
```

What might be the advantage (if any) of writing database access code in this style, compared to working with SQL code as string? Do you agree that using such a framework would be a good idea? Explain.

You can assume that the framework is decently designed and implemented, and not just a very simple example constructed for this exam.

(2–3 paragraphs)

Fill in your answer here

Maximum marks: 15

4 4. HeadBook [a/b/c, 15%]

Consider the *HeadBook* project from Assignment 2 and 3. *(The assignment text is also attached as a resource.)*

***a)* [7%]**

What do you feel would be the main threats against such an application? I.e., what is the *threat model*? Who might attack the application? What can an attacker do? What damage could be done (in terms of *confidentiality*, *integrity*, *availability*)? Are there limits to what an attacker can do? Are there limits to what we can sensibly protect against? **Explain.**

(2–3 paragraphs)

***a)* [4%]**

What are the **most important things you learned** while working on the project, or during the review process?
(2–3 paragraphs)

***b)* [4%]**

What do you feel would be the **most important things** to learn more about to become an effective developer of secure web applications?
(2–3 paragraphs)

Fill in your answer here

Maximum marks: 15

5 5. Exam Injections [a/b/c, 10%]

While making the exam this fall, Anya discovered that Inspera has a number of bugs that make it easy to insert arbitrary code into exam questions. For example, HTML code in multiple choice questions will be inserted into the code without any escaping (see Question 8 for a JavaScript demo).

This vulnerability could be exploited, for example, to run code with administrative privileges when the exam office is checking the exam, or to manipulate a student's exam answers.

The vulnerability is caused by code like the following, which builds the HTML code for the questions and inserts it into the document tree client-side:

```
// Translate internal representation to actual HTML code.
// (Slightly paraphrased to make the syntax more familiar.)
function getAttributesAsString(attrs) {
    return attrs.map(attr => attr.name + '="' + attr.value + '"').join(' ');
}
function getElementAsString(elt) {
    var attrs = getAttributesAsString(elt.attrs);
    var body = getNodeContentAsString(elt.children);
    return '<' + elt.tagName + ' ' + attrs + '>'
        + body + '</' + elt.tagName + '>';
}
// …
mainContent.innerHTML = getElementAsString(questionElt);
```

Let's assume that the client code is full of this sort of code that builds HTML with string concatenation, and the developers are unable to find and close all the injection vulnerabilities.

a) **[3%]**
Assuming an attacker (such as a disgruntled, evil lecturer) is able to use such an exploit to inject arbitrary JavaScript code into the page; what could the Inspera developers do to prevent the injected JavaScript code from actually running? **Explain.**

(1–2 paragraphs)

b) **[4%]**

Let's assume that the developers have successfully prevented attackers from running injected JavaScript code, but it's still possible to insert plain HTML code – like a form or a link, for example.

How could an attacker exploit such a vulnerability? What do we call such an attack?

(1–3 paragraphs)

c) **[3%]**

How would we normally prevent the kind of attack mentioned in *b)*? **Explain.** (Assuming that we're still unable to prevent the code injection from happening in the first place.)

(1–2 paragraphs)
Fill in your answer here

Maximum marks: 10

6 6. CIA [5%]

Confidentiality, Integrity and Availability are often said to be the fundamental concerns or principles of information security – the so called *CIA triad.* All three factors might be important, but in some cases one of them is more important than the others. For example, confidentiality is probably not so important for information which is already public.

For each of *confidentiality*, *integrity*, and *availability*, **give an example** of a system, situation or a type of data where you think that concern is more important than the two others. Remember to motivate your answer.

(~a short paragraph for each)

Fill in your answer here

Maximum marks: 5

7 7. Oblig [40%]

The results from your compulsory exercises will be inserted here.

Maximum marks: 40

8 8. Inspera Exploit [0%]

*This is **not a question**, it's just a demonstration of one of the problems in the "Digital Exam" question. You'll get **no points** for answering it!*

One of the alternatives below contains JavaScript code which will insert the text *"Exam started N minutes ago!"*. (Unless, of course, the actual exam looks different from the preview, which would be another issue.)

Select one alternative:

- Exam started 896093 minutes ago!
- *foo*
- List
- x < 0 && y > 0

Here's what it *should* look like (at 2024-02-21T09:42CET):

Select one alternative:

- List
- Exam started 42 minutes ago!
- x < 0 && y > 0
- *foo*

Maximum marks: 0

Question 3
Attached

1 Code example

```
// Table definition example
public class Enrollment {
    public static Column<Integer> studentId = new IntegerColumn().notNull();
    public static Column<String> courseName = new TextColumn().notNull();
    public static Column<Integer> courseId = new IntegerColumn().notNull();
    public static Column<String> grade = new TextColumn().orNull().length(1).matches("[A-F]");
}

// Example of what the code would look like using the new framework
public class NewVersion {
    /** Return a formatted list of the courses the current user is signed up for. */
    public String viewCourses(String filter) throws SQLException {
        String result = "";
        WhereClause query = Enrollment.studentId.equalTo(currentUser.studentId);
        if (filter != null) // we can combine where clauses with 'and()':
            query = query.and(Enrollment.courseName.like(filter));

        // code is sql-like but uses fluent-style Java method calls
        ResultSet = connection.select().from(Enrollment.class).where(query).execute();

        while (rs.next()) {
            // column knows its type, so we don't need to use getString, getInteger, etc
            result += formatCourse(Enrollment.year.get(rs),
                    Enrollment.courseName.get(rs),
                    Enrollment.grade.get(rs));
        }
        return result;
    }
}

// Example of what the original code looked like
public class OldVersion {
    /** Return a formatted list of the courses the current user is signed up for. */
    public String viewCourses(String filter) throws SQLException {
        String result = "";
        String query = "SELECT * FROM Enrollments WHERE student_id = '"
                + currentUser.studentId + "'";
        if (filter != null) { // add optional course name filter
            query += " AND course_name LIKE '" + filter + "'";
        }
        query += ";";
        ResultSet rs = connection.execute(query);
        while (rs.next()) {
            result += formatCourse(rs.getInteger("year"),
                    rs.getString("course_name"),
                    rs.getString("grade"));
        }
        return result;
    }
}
```

2 Sketch of interfaces for the new framework

```
// Used when we're building queries.
public interface Column<T> {
    // compare to literal value
    WhereClause equalTo(T otherValue);
    // compare to other column
    WhereClause equalTo(Column<T> otherValue);
    // pattern match against value
    WhereClause like(T otherValue);
    // read the value of this column in the given ResultSet
    T get(ResultSet rs);

    // ...
}

// Used when we're defining database tables.
// T is the value type of the column,
// C is the type of the column itself (for correct typing when returning *this*)
public interface ColumnDef<T,C> {
    // require that values are not null
    C notNull();
    // allow null values
    C orNull();
    // specify foreign key
    C foreignKey(Column<T> other);
    // require unique value
    C unique();

    // ...
}

// Some data types might have extra options
public interface TextColumnDef extends ColumnDef<String, TextColumnDef> {
    // define maximum string length
    public TextColumnDef length(int l);
    // define a pattern that values should match
    public TextColumnDef matching(String pattern);

    // ...
}

public class TextColumn implements TextColumnDef {
    // ...
}
```