Lab 3: Interprocedural and Field-Sensitive Taint Analysis

In this lab we implement an interprocedural taint analysis. A taint analysis is useful to detect sensitive data-flows, also known as *taints*, in a program. A classical example of a vulnerability that can be detected by the help of a taint analysis is SQL injection (`https://cwe.mitre.org/data/definitions/89.html`). SQL injection attacks are common in the context of database-backed websites or web applications. Websites commonly access databases by performing SQL queries. A developer has to ensure that user input that is integrated into SQL queries is properly sanitized, i.e. information that comes from a user (or attacker) cannot be used to execute unintended queries on the database. If the input from a user is not properly sanitized, an attacker may access the entire database.

```
String userId = request.getParameter("userId");
Statement st = ...
String query = "SELECT * FROM  User where userId='" + userId + "'";
st.executeQuery(query);
```

Abbildung 1: A simple example of a program code that contains a SQL injection.

Figure 1 shows a minimal example. The developer retrieves the parameter `userId` from a user-controlled request. The parameter is used to construct the SQL query and the query is executed. If a malicious user sets the value of `userId` to `myuser OR 1=1`, the SQL query that is executed is: `SELECT * FROM User where userId=myuser OR 1=1` which will return all users in the table User. In general, a developer cannot trust any user-controlled inputs, (here, the parameters of a request). More information on how to prevent SQL injection can be found here: `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`.

General Instructions

Set up: The code in folder `DECALab3` contains a maven project readily set up. To set up your coding environment:

- Make sure that you have any version of Java running on your machine.
- Import the maven project into your favorite development environment.
- To run the test cases, run the project as a Maven build, with the goals set to "clean test".
- **Important: Make sure that your test cases run via the command line.** To do so, run `mvn clean test` in the `DECALab3` directory.

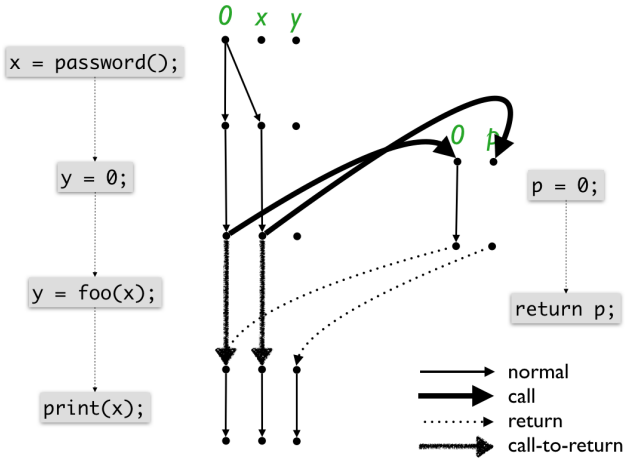
Project: The project contains:

- three `JUnit` test classes in the package `test.exercises` of the folder `src/test/java`. Your goal is to make all of those test cases pass.
- target classes in the packages `target.exercise*.` of the folder `src/test/java`. Those contain SQL injection examples on which the test cases are run.
- three classes containing the flow functions of the three taint analyses you will develop in this lab. They are located in the package `analysis.exercise` of the folder `src/main/java`.

Submission format: Submit on koaLA a file `DECALab3_user1_user2_user3_user4.zip` (with `user{n}` your koaLA logins), containing the project with your solution.

Analysis framework: In this project, we use a different analysis engine than in the previous labs. The analysis solver used here is called IFDS, which you will dive into in upcoming lectures. To do this lab, you only need to know the following notions about IFDS:

- IFDS works on top of Soot, so you will still manipulate Jimple abstractions.
- Unlike Soot, which calls its flowFunction once per Statement, IFDS calls it once per data-flow fact and statement. Let us consider the example of a taint analysis with the statement `c = a;`. Assuming that variables a and b are tainted before the statement, the result of applying the flow function is that variables a, b, and c are tainted after that statement. A Soot-based analysis will call the flow-function once: flowFunction({a ,b}) = $\{a, b, c\}$. IFDS calls it once per element of the input set: ifdsFlowFunction(a) = a, c and ifdsFlowFunction(b) = b. This is done automatically by the IFDS framework, and all you have to do is fill the flow functions, as shown in the package `analysis.exercise` of the folder `src/main/java`.
- ZERO is the original data-flow fact in IFDS. A taint should always be generated from ZERO (e.g. for the statement `a = secret();`, the data-flow fact a will be generated from the ZERO element once it reaches the statement).
- Also unlike Soot which has one flowFunction method, IFDS contains four flowFunction methods which are applied to four different types of statements. *Call flow functions* are applied at call-sites, and ensure that the arguments of the caller are mapped to the correct parameters of the callee before continuing to propagate into the callee. *Return flow functions* are applied at return sites, and map the callee's variables back to the caller's parameters. *Normal flow functions* are applied to all statements that are not call statements, and *call-to-return* flow functions are applied at call statements to handle data-flow facts that are not propagated through the call flow functions. You can see an example in the graph below.



Exercise 1

a) In the first exercise, we implement an interprocedural, *field-insensitive* taint analysis. The analysis will correctly handle parameter assignments of variables at call sites, i.e. when data-flow information is transferred from the call site parameters to formal arguments of the callees. For this exercise, data-flows to fields (and escapes to the heap) are ignored. Only changes within the file `Exercise1FlowFunctions.java` are required. The class contains the implementation of the flow functions. Do not change any other file for this exercise.

Start by the implementation of the most simple test case for SQL injection. The `JUnit` test `directSQLInjection` in `Exercise1Test` analyses the target code class `DirectSQLInjection`. Change the flow functions such that: *(1 point)*

- When the data-flow fact `ZERO` reaches a call statement `x = getParameter(...)`, generate a data-flow fact that represents `x`.
- The test case `directSQLInjection` must pass.

To generate the data-flow fact, use the appropriate constructor of `DataFlowFact`.

b) Modify the method `getNormalFlowFunction` in `Exercise1FlowFunctions.java` to support assign statements of the form `x = y`: *(1 point)*

- When a local data-flow fact `y` reaches the statement, generate a local data-flow fact for `x` and add it into the `out` set.
- The test case `assignmentSQLInjection` must pass.

c) Modify the method `getCallFlowFunction` in `Exercise1FlowFunctions.java` to handle interprocedural data-flows: *(1 point)*

- When a taint `x` reaches a call site of form `foo(x)`, map the variable `x` to the respective parameter local of the callee method. The code in method `modelStringOperations` may help you as it implements a similar functionality.
- The test case `interproceduralSQLInjection` must pass.

d) Make sure that the test case `noSQLInjection` also passes. *(1 point)*

Exercise 2

We now make the analysis *field-based*, i.e. we enable it to support flows through fields. You can reuse all of the code you have implemented in `Exercise1FlowFunctions`, copy your change into class `Exercise2FlowFunctions`. For this exercise you are only allowed to make changes within this class. Adjust the flow-functions in `Exercise2FlowFunctions` such that: *(3 points)*

- The analysis supports field store statements of the form `x.f = y`: generate a data-flow fact that represents the field `f` if `y` is tainted. The class `DataFlowFact` provides a constructor that takes a `SootField` as its single argument. Use this constructor for the field-based analysis in this exercise as it ignores the base variable.

- The analysis only detects the data-flows if we also model field load statements, i.e. statements of the form `y = x.f`. Add the respective functionality.
- Do not forget to model fields flowing at call sites to the callees, i.e. extend `getCallFlowFunction` appropriately.
- All test cases of `Exercise2Test.java` must pass.

Exercise 3

Consider the code of class `FieldNoSQLInjection`. The field-based static analysis discovers an SQL injection, but there is actually none in the code. The field-based analysis reports one warning for the test case `fieldBasedImpreciseTest` in `Exercise2Test`. A precise analysis does not report any warnings.

In this exercise, we want to modify the field-based analysis to make it more precise and create a *field-sensitive* analysis. Implement your flow functions in class `Exercise3FlowFunctions`. You can reuse most of the code implemented for **Exercise 2**. *(3 points)*

- Modify the handling of field store and load statements in the analysis. Generate a data-flow fact `x.f` for a statement `x.f = y`, when `y` is tainted. Ensure this time that the data-flow fact also models the base variable of the field store, i.e., `x`.
- Correspondingly adjust the handling of field load statements.
- All test cases of `Exercise3Test.java` must pass.

Exercise 4

The data-flow fact model in `DataFlowFact.java` only models a single field access. This limitation is enough to make the analysis miss injections. Craft a target code which exploits this limitation: *(0 points)*

- Implement a target code example in `UnsoundExample.java` on which the field-sensitive analysis of **Exercise 3** is unsound, i.e. it does not report a warning but should produce one. The data must flow through at least two fields.
- To guide you finding the target code, the class `UnsoundExample` contains a comment with a list of statements you should use.
- If the test case `Exercise4Test` passes, you have found the right target code.