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 typical 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.

Listing 1: A simple example of a program code that is vulnerable to SQL injections.

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

Listing 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`.

Your Options

Please make your choice whether you wish to use Java/SootUp or C++/PhASAR.

- If you wish to use Java/SootUp, you have to solve exercise 1 - 3.
- If you wish to instead use C++/PhASAR, you have to solve exercise 4 and 5.

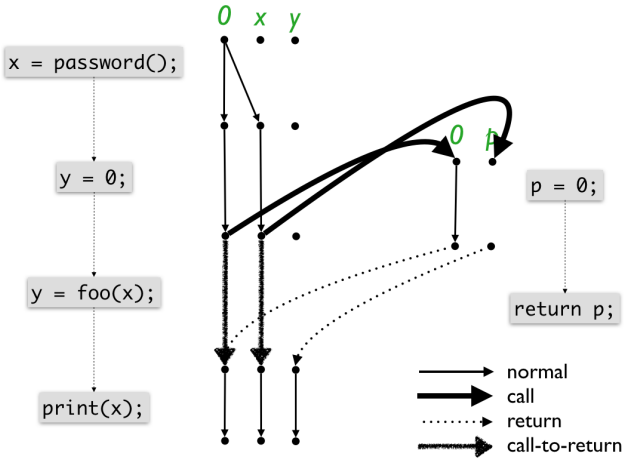
Submission format: Submit on PANDA a file that follows the naming scheme `group_A_language_lab1.zip` (replace `A` with your own group and replace `language` by `java` or `cpp`) that contains the project with your solution.

Analysis framework: In this project, we use a different analysis engine than in the previous labs: IFDS. To do this lab, you only need to know the following notions about IFDS:

- IFDS calls its flow function factories once per statement. The concrete flow function that is constructed for each statement is then queried per individual data-flow fact that holds at the statement currently under analysis. 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. The Monotone Framework would call the respective flow function once: flowFunction({a ,b}) = $\{a, b, c\}$. IFDS, on the other hand, 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.

- The tautological Λ (or ZERO) fact is the original data-flow fact in IFDS. A taint should always be generated from Λ (e.g. for the statement `a = secret();`, the data-flow fact a will be generated from the Λ element once it reaches the statement).
- IFDS contains four flow function factories which are queried according to the current statement under analysis. *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 non-call and non-return 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.



General Instructions

Set up: The code in the directory `DECALab3-SootUp` 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-SootUp` 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`.

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 implementing 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.

PhASAR-based Lab

General Instructions

Set up: The code in the directory `DECALab3-Phasar` contains a CMake project readily set up. Please refer to the project's `README.md` on how to build and interact with the project.

Project:

The project contains:

- A main program `taint-analysis.cpp` that you can use to get a feeling for the implementation.
- The actual analysis implementation that you can find in `include/Lab03/IFDSTaintAnalysis.h` and `lib/IFDSTaintAnalysis.cpp`, respectively. We added `TODO` markers in the comments such that you can easily spot the places you need to touch. Just `grep` for the todo markers.
- Unit tests that we implemented using the googletest framework. You find the unit tests in `unittests/IFDSTaintAnalysisTest.cpp`. The unit tests can be automatically executed by running `ctest` in your build directory (please refer to `README.md`).
- Target programs that are used to evaluate your analysis implementation. You can find the target programs in `target/`.
- Once you compiled the entire project, you will find the LLVM IR for the test programs in `build/target/`. Please take a careful look at the IR that is generated as this is the code your analysis is eventually run on.

Exercise 4

Implement a field-insensitive taint analysis using IFDS. As data type for your data-flow domain you can use `D = d_t = const llvm::Value*`; check out the default setup of the analysis domain: `TaintAnalysisDomain`.

We recommend to use the following strategy:

- Adjust the implementation of `initialSeeds()` such that the formal parameters of the `main()` function (in the target code) are considered to be tainted as those contain potentially malicious, unchecked user input.
- Next, implement the `getNormalFlowFunction()` flow function factory that is queried for all intra-procedural, non-call, non-return instructions. PhASAR already provides ready-to-use flow-function templates: You may want to take a look at the class `FlowFunctionTemplates`[1], which is a base class of the `IFDSTaintAnalysis` that you implement.
  Note: When handling the `store` instruction, you may need to generate aliases of the pointer operand.

[1] `https://github.com/secure-software-engineering/phasar/blob/development/include/phasar/DataFlow/IfdsIde/FlowFunctions.h#L145`

- You may wish to implement some *special* semantics for the `std::string` type's special member functions. For instance, you can model `std::string`'s constructor(s) such that you do not need to analyze those. Use the `getSummaryFlowFunction()` function for that. If it returns a non-null flow-function, the solver will not invoke `getCallFlowFunction()`.
- Implement the `getCallToRetFlowFunction()` that propagates all data-flow facts that are not involved in the call alongside the call site. This function usually propagates all data-flow facts that are not involved in the call as identity. All data-flow facts that are used as pointer or reference argument in the call are usually killed as these flow facts are handled by the `getCallFlowFunction()` and `getRetFlowFunction()` functions. You may want to take a look at the helper function `mapFactsAlongsideCallSite()` from PhASAR.
- Implement the `getCallFlowFunction()` function that maps the actual parameters at the call site to the formal parameters of the potential callee target. You may want to take a look at the helper function `mapFactsToCallee()` from PhASAR.
- Implement the `getRetFlowFunction()` function that maps any formal (pointer or reference) parameters at the exit instruction, i.e. the return instruction, back to the corresponding actual parameters at the respective call site. The implementation must also transfer any potential return flows to the left-hand side variable of the respective call site.
- You may also wish to craft a sensible implementation for `emitTextReport()` such that you can report your analysis results in a meaningful manner.

Please also consult PhASAR's documentation on the four flow function factories that can be found in `include/phasar/DataFlow/IfdsIde/FlowFunctions.h`.

Hint: You may modify the implementation of the function `emitTextReport()` to customize, how the `taint-analysis` tool will display the analysis results.

(6 points)

Exercise 5

Make your analysis field-sensitive such that all unit tests pass.

To make your taint analysis field-sensitive you need to come up with a model to represent fields. We recommend to model a custom type `FSFact` that represents the data-flow facts in a field-sensitive manner. (Hint: have your custom type store the base variable and the potential offset(s) using the respective `llvm::GetElementPtrInst` (GEP) instructions. You may also want to compute actual byte-offsets using `accumulateConstantOffset()` on the GEP.) Once you designed your type to represent the flow facts in a field-sensitive manner, adjust your analysis to use `D = d_t = FSFact` such that is able to distinguish fields. Note, that your flow-fact type needs to be comparable (`==` and `<`), and hashable (via `std::hash`). To make your `FSFact` type printable within the framework, implement a function `std::string DToString(const FSFact &)` within the same namespace, such that it can be found through ADL.

Adjust the implementations of the flow function factories. Depending on your design and implementation of `FSFact` you may need to adjust the `isZeroValue()` and `initialSeeds()` functions,

too.

(4 points)