Contents

1. Implementation

This chapter presents the implementation of our project. Since our project involved both extending Weverca analyzer and contributing to the Eclipse IDE, the chapter is divided into three separate parts. The first one explains implementation of taint analysis in Weverca analyzer, the second one depicts some changes that had to be made in Weverca analyzer in order to implement the IDE integration and the last one briefly describes implementation of Eclipse plug-ins and their connection to the analyzer.

1.1 Taint Analysis

The basic idea of taint analysis is to keep track of values that originate in user input. As a result it is able to determine variables which are potentially influenced by outside input (therefore have to be considered tainted) and warn the developer when any of these variables is used to execute dangerous commands (such as commands to an SQL database).

The most elementary version of taint analysis only propagates boolean values indicating whether the variable is tainted or not. This approach does not fit our intentions for we also want to determine all the potential sequences of assignments from the source to the sink and distinguish between various taint flags since tainted variable may be sanitized (for example for usage in SQL commands). This leads to a need for propagating the actual flows and individual taint flags. Another useful information is whether all the possible paths lead to a tainted value or there exists at least one safe path. And finally, even though they do not originate in user input, we also decided to propagate null and undefined values, for it can indicate bugs in input filtering and moreover it does not make any sense to use variables with these values to execute a command.

Taint analysis described in this section is implemented in the Weverca.Taint project. Unit tests for taint analysis are implemented in file TaintAnalysisTest.cs, which is part of Weverca.AnalysisFramework.UnitTest project. Abstract classes that our taint analysis extends are from Weverca.AnalysisFramework project.

The taint analysis is implemented as a `TaintForwardAnalysis` class which is derived from `NextPhaseAnalysis` abstract class described in section ??. All the `TaintForwardAnalysis` needed to implement was intialization of tainted variables such as _POST, _GET, _REQUEST etc. The following subsections describe the taint analyser, which was implemented using the class `TaintAnalyzer`—

implementation of the class `NextPhaseAnalyzer`.

1.1.1 Taint Flow Representation

Taint flow representation is called `TaintInfo` and it is always associated with a specific program point. Each instance of `TaintInfo` contains multiple fields:

1. A program point the instance is associated with.
2. A list of all possible taint flows that lead to the associated program point. This list contains pairs of `TaintInfo` information and `VariableIdentifier` which identifies the variable that the taint came from if such a variable exists.
3. An indicator of three possible taint flags—HTML taint, SQL taint and file path taint.
4. An indicator of priority for each taint type. The priority is high if all the flows that flow into the program point have the corresponding taint flag.
5. A null value indicator for determining flows of null or undefined value.
6. A general taint indicator which determines whether there exists a flow from user input. This information is independent of the individual taint flags which could have been sanitized as described in section 1.1.2.

1.1.2 Taint Analyzer

The purpose of the `TaintAnalyzer` class is to define behaviour for visiting various types of program points. In this case it means defining an appropriate taint flow propagation and handling. The taint information is stored in program point's InfoLevel input set and output set, all already described in section **??**. In general, there are only three operations that can be done with a taint information:

1. **Merging and extending taint information**
   This involves merging all the taint information entering the program point into one information extended by the current program point. The entering taint information is collected differently for each program point, however the actual merging is always done the same way. At first a new, initially untainted, taint information is created for the current program point. Afterwards, all the entering taint information is processed. One after another all the instances of `TaintInfo` are stored in the list of possible taint flows, and their taint information is propagated—if any of them is tainted, has a

possible null value or has a specific taint flag, the information is propagated to the result. On the other side, if any information does not contain a high priority indicator for a flag, it is removed from the new `TaintInfo` instance too. In the end, the newly created `TaintInfo` instance is produced.

2. **Reporting the taint information**
   Reporting the taint information means merging the entering information and possibly creating a warning out of it. A special type of warning - `AnalysisTaintWarning` was implemented for the purpose of taint analysis. It is derived from `AnalysisWarning` and contains additional taint flow information. All the created warnings are stored in a list accessible from `TaintForwardAnalysis`. Warnings can be optionally based on a specific taint flag, for example in comparison to `eval` statement which reports any flow that originates in user input, `echo()` function only reports flows that have a HTML taint indicator set to true. Independently of the taint flags, when reporting a taint information with null value, a warning with null flows is created too.

3. **Sanitizing the taint information**
   The user input always has all the possible taint flags. Sanitizing the taint information means removing one or more taint flags when a special sanitizing function is called. The list of sanitizing functions contains for example `htmlentities()` function which removes the HTML taint flag, or `md5()` function which removes all of them.

Both reporting and sanitizing requires the taint information to be merged first. Merging has to be done because of multiple taint information entering the program point, which is caused either by multiple variables contributing their taint information (for example multiple function arguments) or by a single variable having multiple `TaintInfo` values (caused for example by merging information at join points of conditional branches).

As already mentioned, different program point types have different behaviour for visiting implemented in `TaintAnalyzer`. To demonstrate how it works we will describe a few examples:

- `AssignPoint`
  This point represents an assign expression and contains two operands—right operand and left operand. To propagate the taint, a right operand taint information must be extended by this program point and stored in the left operand's memory. To achieve this independently of the right operand

type (it may be, e.g., a variable, an expression, and a function call) the right operand taint information is merged and then extended.

- `EchoStmtPoint`

  `EchoStmtPoint` represents an echo statement which obviously requires reporting, for HTML tainted data should never enter the browser. At first, the taint information is merged from all the parameters of echo statement. If the merged taint information has a HTML taint flag or possible null value, a warning is created.

- `NativeAnalyzerPoint`

  This point represents a native method call. The taint information of all the arguments needs to be collected and merged. Then it is checked whether the method is a sanitizer or a reporting function, if yes, the merged taint information is appropriately handled. Finally the resulting taint information is stored as a method return value, to be possibly processed later as for example `AssignPoint`'s right operand.

After a program point is processed, the `NextPhaseAnalysis` checks whether the result of computation adds any new information to the information computed the last time the program point was processed. If it is, all the following points are added to the worklist to be processed. This brings up a question of infinite loops. In case of a cycle in the source code, it can easily happen that the taint flows will be extended by the same set of program points over and over again. The way to avoid this situation is described in the following section.

1.1.3 Avoiding Infinite Loops

The decision whether to process a program point again is based on the fact whether its value adds any new information to the value computed when the node was processed the last time. Since this value is our `TaintInfo` information, to avoid an infinite loop it is only necessary to properly define equality for `TaintInfo`.

The implicit equality which compares the actual `TaintInfo` instances is obviously improper, for there is a new instance of `TaintInfo` created anytime merging takes place. A better option is to compare equality of all the `TaintInfo` fields. However, this solution is also insufficient. Consider this piece of PHP code:

```
$a = $_POST['a'];
while(true){
        $a=$a.$a;
```

```
}
```

Inside of the cycle the information about previous flow represented by pairs of `TaintInfo` and `VariableIdentifier` will never stabilize, since the flow will be always extended by the same program point and will always be longer than the previous one.

The solution to this problem is quite simple—the defined equality should require all the `TaintInfo` fields to be equal except for the list of possible taint flows. The equality condition for this list should be weaker—it is only necessary that the taint information is collected from the same set of program points. When there is an infinite loop in taint propagation, only program points which were already present in previous iteration are added to the taint flow from the previous iteration. The set of program points in the taint flow is thus equal to previous iteration and the computation stabilizes.

1.2 Weverca Analyzer Adjustments

Weverca analyzer is not only a library, it also contains a console application project which can be executed with parameters defining whether to run static analysis or compute metric information. For our purposes we added an additional parameter option `-cmide` used for IDE integration. This parameter requires another parameters to determine the action, file or files to analyze and optionally other parameters depending on the action type.

There are three basic actions: getting the metric information, running static analysis and getting the construct occurrences. The last action can be done in two possible ways - either a PHP file to get the occurrences from or a file containing all the file paths to get the occurrences from is provided. All the information provided by `-cmide` option is returned in the form that is adjusted for parsing by plugins.

1.3 Eclipse Plug-ins

The main part of our project was integration of PHP code analysis into Eclipse IDE. First of all this involves implementation of the analysis launch. This is either scheduled to be done automatically or by direct or indirect user requests. An indirect analysis request is for example selecting a file in a file explorer. After the analysis is done, the output has to be parsed into a data structure that can be processed and displayed to the user in various forms, usually as a clickable view

or code highlight. In case of a large analysis, this parsing is quite time consuming and the resulting structure may take up a lot of space.

1.3.1 Requirements

The project does not provide an Eclipse perspective, its implementation requires use of PHP perspective provided by PHP Development Tools (PDT). Additionally, most of the functionality is also available for the basic Resource perspective. The PDT plug-in is necessary for analyzing the files selected in PHP explorer which is a view defined by this plug-in and it is also needed to only provide possibility of static analysis when PHP editor is opened.

1.3.2 Architecture

This part of our project is composed of seven individual plug-ins. In general they can work independently, however there are two exceptions. First, all the plug-ins depend on the `Common` plug-in. Second, both `StaticAnalysisWarnings` and `StaticAnalysisVariables` need `StaticAnalysis` plug-in to be launched. Yet the dependency is the other way around, for the `StaticAnalysis` plug-in requires `StaticAnalysisWarnings` and `StaticAnalysisVariables` plug-ins to execute them. The dependencies can be seen in figure 1.1.

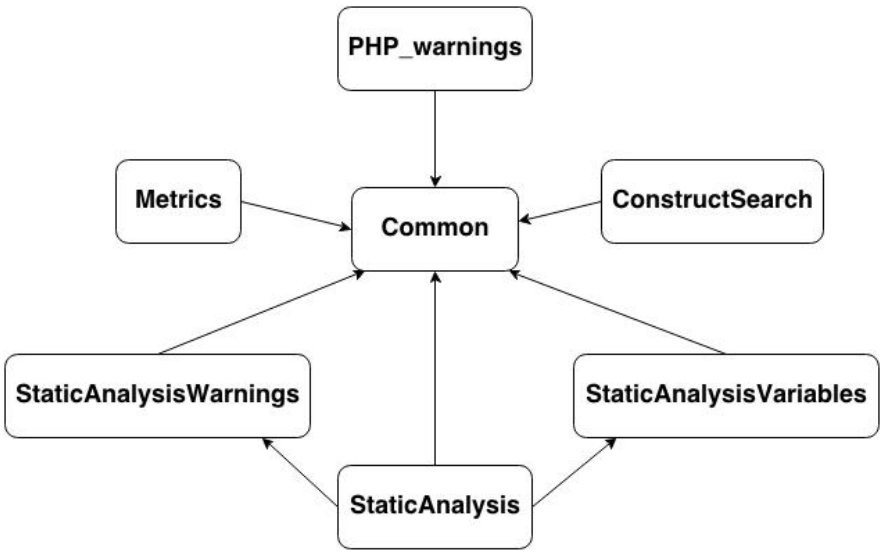

Figure 1.1: Dependencies between plug-ins. The arrows point from a plug-in to its dependencies.

1.3.3 Plug-in Common

This plug-in contains classes and methods that are used in multiple plug-ins and also some images used as icons. The most important functionality it provides is a connection to Weverca analyzer, implemented in the class `Runner`. The connection is realized using `ProcessBuilder` which creates operating system processes and allows reading the analyzer's standard output as an input stream. Any time a connection of a plug-in to analyzer is presented in this text, it implicitly means connection using the `Runner`.

The only contribution of this plug-in to the Eclipse workbench is a preference page that allows to define a path to Weverca analyzer. Some of the other functionality of this plug-in is for instance getting the file paths of files selected in a specific file explorer, providing icons or putting focus on a specified position in an editor.

Figure 1.2: Connection to Weverca analyzer

1.3.4 Plug-in Metrics

The `Metrics` plug-in provides two view parts, `ViewAggregated` and `ViewSimple`, which are able to work both independently and dependently. If only one of them is open, it uses an implementation of `ISelectionListener` to listen for selection changes in file explorer—either `PHP Explorer` or `Project Explorer`. After the selection change a `MetricsParser` is created which calls `Runner` with selected files and directories as Weverca analyzer parameters, and parses the information. If both view parts are open simultaneously, only the `ViewAggregated` creates a `MetricsParser` and provides the `ViewSimple` with all necessary information.

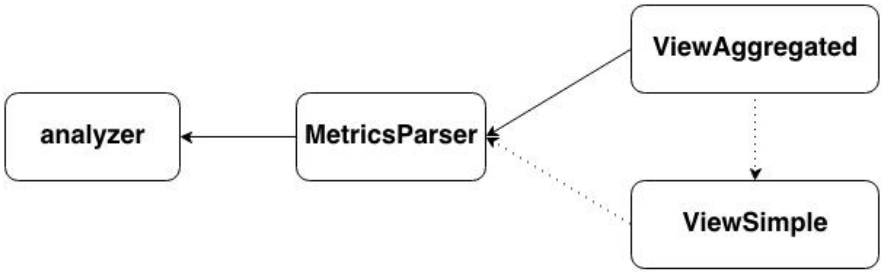

Figure 1.3: Metric information flow

The `MetricsParser` parses the metric information from analyzer into two

different metric information representations. The first one, `MetricInformation`, stores metric information of a single file or directory. The second representation, `AggregatedMetricInformation`, stores a recursively merged metric information of all the files that were selected. This information is then requested by the view parts and displayed in tables located inside of them.

1.3.5 Plug-in ConstructSearch

This plug-in contributes to a search page extension. The search page allows defining which constructs to search for, either in files that comes from the file explorer selection or in files from the whole workspace. The search page uses a `ConstructParser` class to get the construct occurrences from analyzer and parse them. When the result is available, it opens a search result view which shows the construct occurrences in a form of `TreeViewer`, that is provided by JFace. The `TreeViewer` hierarchy can be seen in figure 1.4. The `TreeFile` and `TreeFolder` objects represent the file structure and each `TreeConstruct` represents an occurrence of a construct. The occurrences are also highlighted in the editors. The highlight implementation uses Eclipse's `Annotations` managed by `IAnnotationModel`.

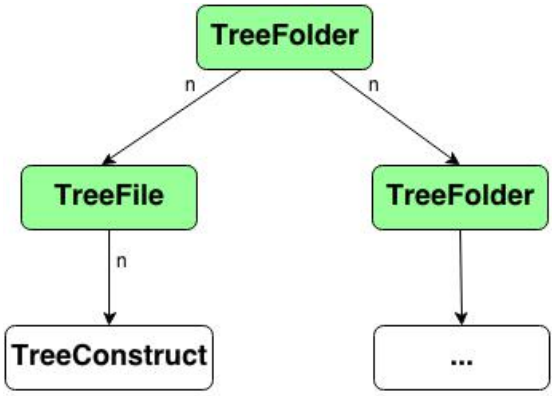

Figure 1.4: Hierarchy of the TreeViewer in the search result view. Green objects represent an object that can possibly be the root of the tree. The 'n' represents the number of possible descendants of the specific type, which is a non-negative integer.

1.3.6 Plug-in PHP_warnings

The PHP_warnings plug-in marks the construct occurrences using the same tools as the `ConstructSearch` plug-in. It contributes to the preference page extension, allowing developer to choose the constructs that will raise a warning. The way this plug-in works is that it monitors editor changes and has an action scheduled

to be executed every second using a `ScheduledExecutorService`. This action checks whether there were any changes made in an editor and if yes, the analyzer is called and the highlight is updated according to the new construct occurrences. This plug-in has to deal with a problem of dirty editors, because there is a need to analyze a file that is not saved. As a solution a temporary file with editor content is created and analyzed.

1.3.7 Plug-in StaticAnalysis

When the static analysis is called, either from project menu or from file explorer pop-up menu, this plug-in calls analyzer and pushes the result to `StaticAnalysisWarnings` and `StaticAnalysisVariables` plug-ins to process it.

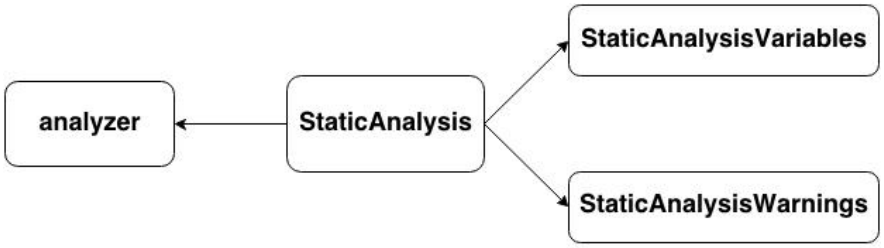

Figure 1.5: Static analysis execution

1.3.8 Plug-in StaticAnalysisWarnings

This plug-in is only executed when called from `StaticAnalysis` plug-in and its input is the static analysis result extended by taint analysis result from Weverca analyzer. This input is parsed into individual warnings using a `WarningsParser`. A warning can potentially contain one or more call stacks and, in case of a security warning, it also contains one or more taint flows. Each warning has a priority indicator that is high only if it is a security warning with all possible flows tainted. All this information is visualized as a `TreeViewer` inside of a `WarningsView` and the warnings are highlighted in the editors. The hierarchy of the `TreeViewer` can be seen in figure 1.6. `Warning` type represents a warning, which can optionally contain call stacks represented by `Call` and taint flows represented by `FlowString`.

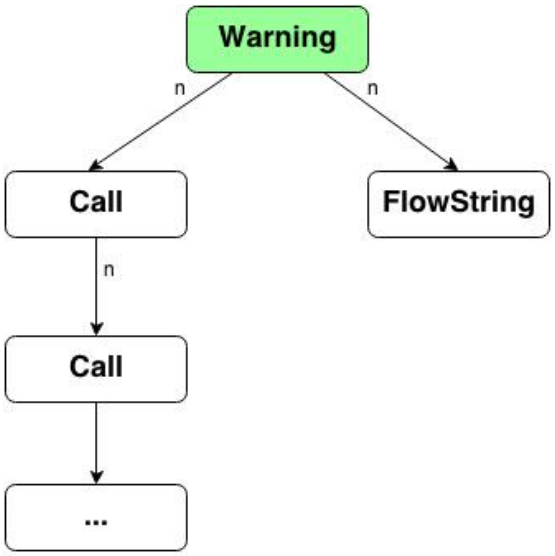

Figure 1.6: Hierarchy of the TreeViewer in the WarningsView. Green objects represent an object that can possibly be the root of the tree. The 'n' represents the fact that the number of possible descendants is a non-negative integer

The `TreeViewer` has a listener (an implementation of `IDoubleClickListener`) registered, which listens for double click events. When a taint flow is double clicked, this listener opens it in another view, called `TaintFlowView`. The taint flow is also showed in the form of `TreeViewer`, with hierarchy demonstrated in figure 1.7. If the flow does not represent merged flows, it only comprises of one layer, in case of merged flows, this hierarchy may be split into multiple layers. The first layer represents the common beginning of the flows with their source, the second layer represents their middles that are different and the third layer represents the common ending of the flows with their sink. Any of these layers may be omitted, since, for example, the merged flows do not have to have a common source. Each layer comprises of a list of objects of `FlowPoint` and `Resource` type. Additionally, if multiple layers are present, there are two special types of `FlowPoint`—`FirstFlowPoint` and `SplitPoint` that serve for a tree representation.

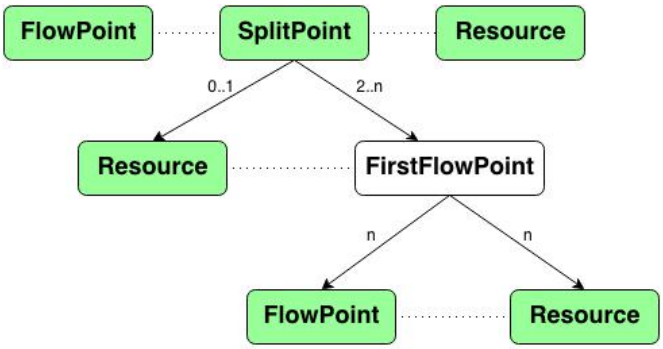

Figure 1.7: Hierarchy of the TreeViewer in the TaintFlowView. Green objects represent an object that can possibly be the root of the tree. The 'n' or numbers represent the number of possible descendants of the specific type ('n' stands for a non-negative integer). Dotted lines represent the fact, that objects in the same level of the hierarchy form a list structure.

1.3.9 Plug-in StaticAnalysisVariables

This plug-in is executed after the `StaticAnalysisWarnings` and handles the same input information. The analyzer provides representation of MemoryLevel input and output set for each line, containing information about all the variables. These representations are parsed by `VariablesParser` into a structure which is then presented to the developer. Besides presenting the content information of variables, this plug-in is also responsible for dealing with dead code. If dead code is detected, a `DeadCodeView` is opened with a `TreeViewer` with simple hierarchy of unreachable program points and optionally their call stacks.

The content of variables is displayed in a view called `VariablesView`. Once again, this view contains a `TreeViewer` that shows information about input or output set of one line in the code. The structure of this `TreeViewer`, which can be seen in figure 1.8, is much more complex than the previous ones, for it has to systematically present a lot of information. `ProgramPoint` represents information about one line of a code. `Context` represents a context—a line of code can have multiple contexts if it is, for example, a part of a function body. In this case, different function calls can lead to different information. Therefore `Context` can optionally contain a Call which represents the call stack. It also contains a `VariableType` which depicts three types of variables - local variables, global variables and aliases. The first two types contain a list of `Variable` objects and the alias type contains a list of `Alias` objects. An `Alias` may possess a must and may `AliasType` which represent the aliases that must be true and the ones that may be true. Each `AliasType` contains a list of aliased variables. `Variable` holds three different kinds of information. The first, represented by `Value` contains the

variable type and value, the second is a taint information represented by `Taint` and its children `TaintValue`. The last information that `Variable` may hold is a list of its fields as `Variable` objects. This information is only available in case of an array or an object.

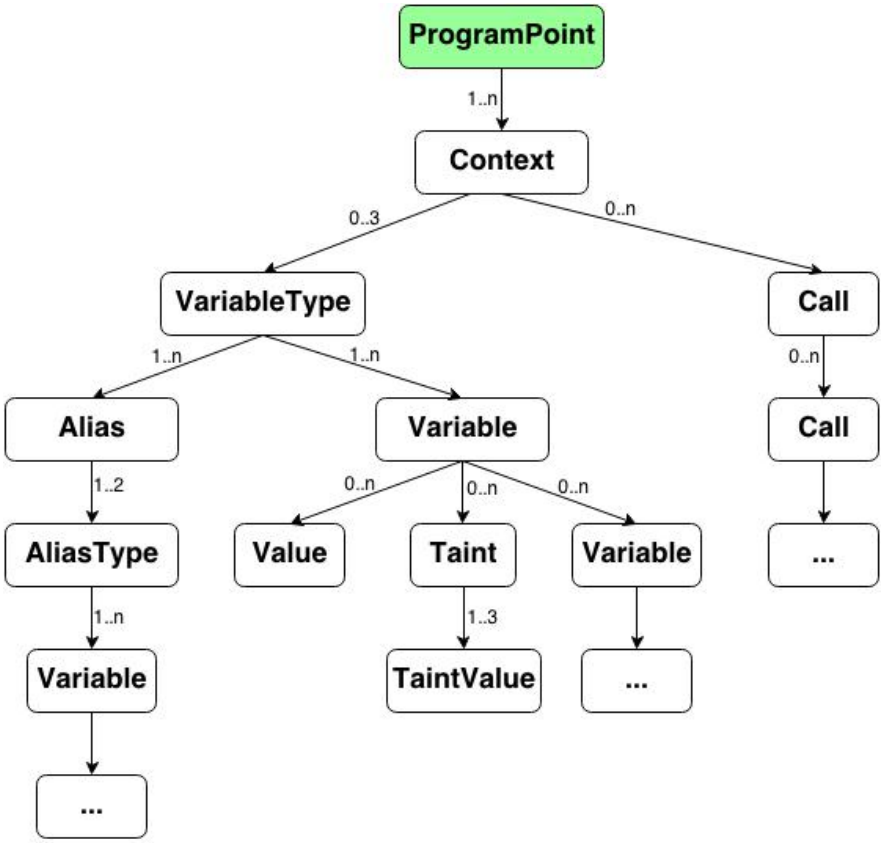

Figure 1.8: Hierarchy of the TreeViewer in the VariablesView. Green objects represent an object that can possibly be the root of the tree. The 'n' or the numbers beside lines represent the number of possible descendants of the specific type ('n' stands for a non-negative integer).