Contents

1. User Documentation

In this chapter we present a set of plug-ins for Eclipse IDE that provide PHP code analysis. The plug-ins use external project called Weverca to analyze the PHP code both syntactically and statically. As a result, our plug-ins are capable of displaying code metrics, finding specific language construct, detecting unreachable code, showing the set of potential variable values, pointing out the code flaws or visualizing potentially dangerous flows from user input into database or other dangerous sinks. This information is not only useful for source code quality maintenance but also for developing secure web applications.

It is not easy to create a secure application and is even impossible to manually detect all the security vulnerabilities. Hence it is useful to use automated tool that makes this less time consuming and more reliable. Our plug-ins are able to detect some security flaws in the code and help to detect many others. In both cases it is necessary to review the analysis results manually, however the plug-ins provide a nice and simple output that can be reviewed easily and quickly.

This chapter describes the functionality of our plug-ins and describes the output of the analyses.

1.1 Requirements

The following list contains the required software:

- Weverca analyzer
- .NET 4.5 or later/Mono 3 or later
- Eclipse version 4.1 or later
- JRE 1.7 (Java 7) or later
- partially required Eclipse PDT (PHP Development Tools) plug-in

The plug-ins were developed and tested using the following configuration:

Operating system:	Windows 7 and Windows 8.1
Eclipse:	version 4.3.2
Java:	version 7
PDT:	version 3.0.1

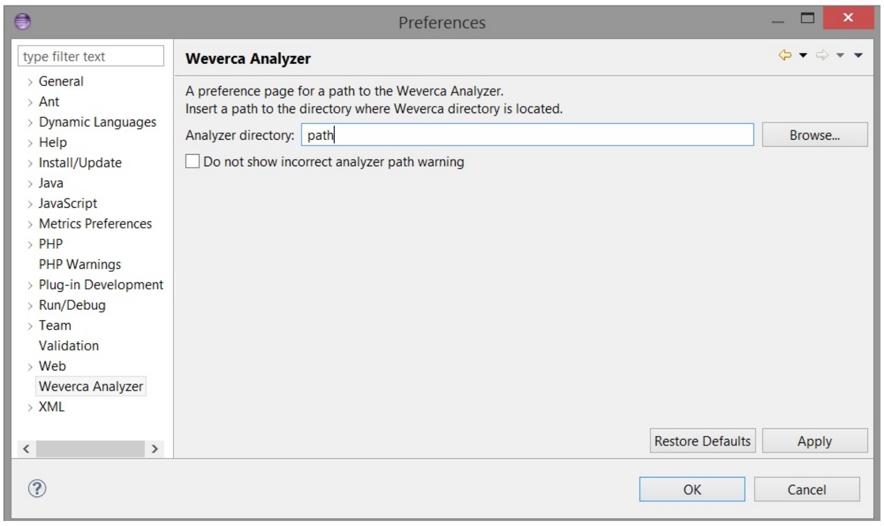
1.2 Installation

The attached CD contains an Installation folder with a feature project called PHP.analysis.plugin. In Eclipse, feature projects are easily installed using the Install page. This can be found in the main toolbar `Help -> Install New Software...`. This page requires a site to work with, which is added by clicking the Add button. Here, the Installation folder should be added as a Local site. Eclipse will search for a feature project in this location and after selecting the project to install, the installation manager will guide the user through the installation process.

1.3 Preferences

The most important setting is increasing the size of available memory for Eclipse. This can be done in Eclipse.ini file by setting the `-Xmx` and `-XX:MaxPermSize` arguments. The necessary size depends on the size of analyzed projects—1024m should be sufficient, however it is necessary to enlarge it, if analysis is not working properly.

Another preference is the path to Weverca analyzer. The default location is the same directory as the plug-ins are located in, however this can be changed in a preference page in the main toolbar `Window -> Preferences -> Weverca Analyzer`. If the path is not correct, the plug-ins raise a warning. The preference page also provides a possibility of hiding this warning.


Weverca Analyzer

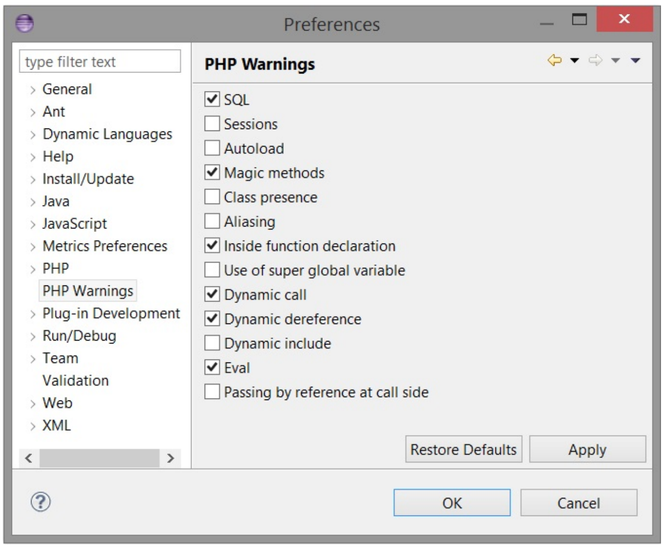
A preference page for a path to the Weverca Analyzer.
Insert a path to the directory where Weverca directory is located.

Analyzer directory: path

☐ Do not show incorrect analyzer path warning

Figure 1.1: Preference page for defining the path to Weverca analyzer

The user has a possibility to define which constructs should report an on-the-fly warning. This can be done in a preference page that can be found in the main toolbar: `Window -> Preferences -> PHP Warnings`. The preference page provides an option to select from 13 different constructs.

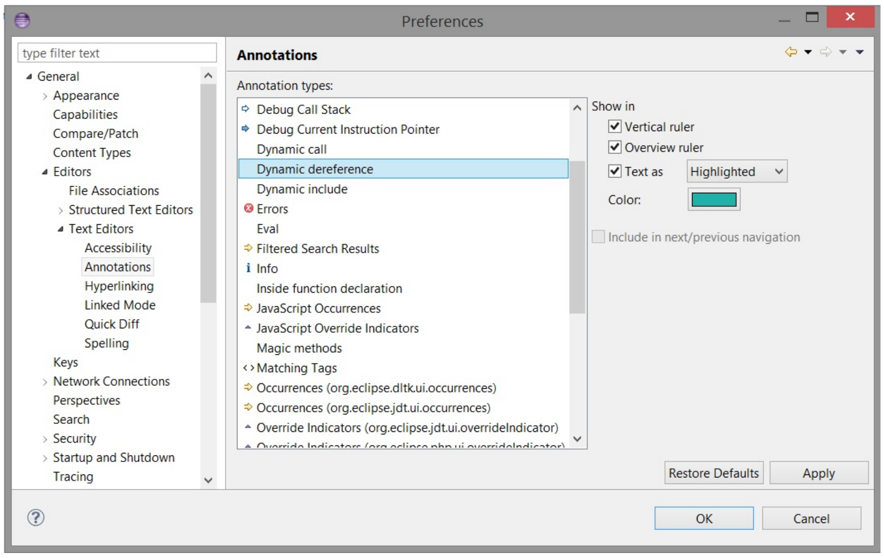


PHP Warnings

- [✓] SQL
- [ ] Sessions
- [ ] Autoload
- [✓] Magic methods
- [ ] Class presence
- [ ] Aliasing
- [✓] Inside function declaration
- [ ] Use of super global variable
- [✓] Dynamic call
- [✓] Dynamic dereference
- [ ] Dynamic include
- [✓] Eval
- [ ] Passing by reference at call side

Figure 1.2: Preference page for defining the constructs to raise a warning

Besides that, the user can also define how to highlight the individual construct search results (SQL, Class, Sessions, Autoload, Magic methods, Aliasing, Inside function declaration, Super global variable, Dynamic call, Dynamic dereference, Dynamic include, Eval, Passing by reference at call site) and PHP construct warnings (PHP Construct Warning). The developer can decide whether to highlight or only show a squiggly line, set the highlight color and choose whether to show this annotation in a vertical and overview ruler. This can be all defined in `Window -> Preferences -> General -> Editors -> Text Editors -> Annotations`.



Annotations

Annotation types:

- Debug Call Stack
- Debug Current Instruction Pointer
- Dynamic call
- Dynamic dereference
- Dynamic include
- Errors
- Eval
- Filtered Search Results
- Info
- Inside function declaration
- JavaScript Occurrences
- JavaScript Override Indicators
- Magic methods
- Matching Tags
- Occurrences (org.eclipse.dltk.ui.occurrences)
- Occurrences (org.eclipse.jdt.ui.occurrences)
- Override Indicators (org.eclipse.jdt.ui.overrideIndicator)

Show in

- [✓] Vertical ruler
- [✓] Overview ruler
- [✓] Text as: Highlighted

Color: 🟦

- [ ] Include in next/previous navigation

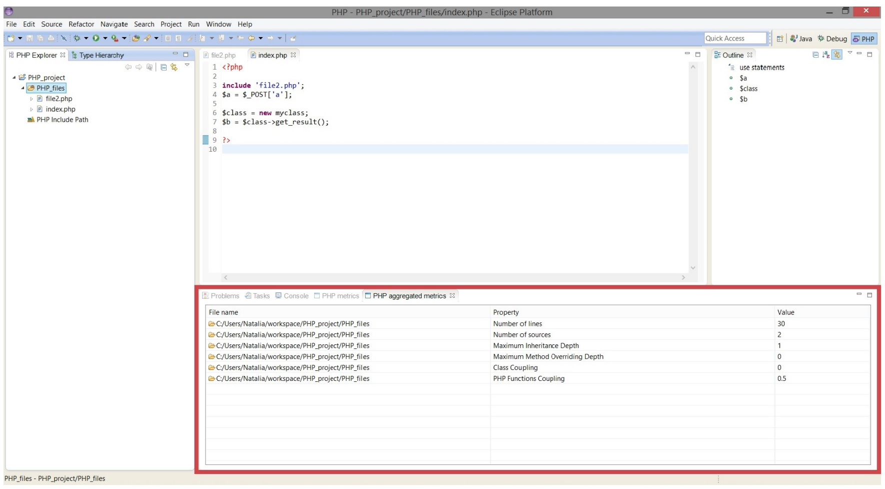
Figure 1.3: Preference page for annotations

1.4 Usage

After the plug-ins are installed, they are ready to be used. First of all it may be necessary to define the correct path to the Weverca analyzer. This can be done in a preference page as described in section 1.3. The plug-ins cover three main areas - metrics computation, potentially dangerous constructs visualization and static analysis. All three areas are covered in the following subsections. The first two are executed quite fast (naturally the actual time depends on the amount of analyzed code) static analysis may take a notably longer time, however it provides a lot of useful information. Most of the computations, especially those that may take a long time, are done in the background.

1.4.1 Code Metrics

There are two available views for displaying the metric information—PHP metrics and PHP aggregated metrics. To get the metric information it is necessary that at least one of the metrics views is open. To find these views and all the other views provided by our plug-ins go to `Window -> Show View -> Other... -> PHP Analysis`. Besides these views, a Project Explorer view (Resource perspective) or PHP Explorer view (PHP perspective) must be open too. In order to compute the metrics, the user have to select one or more files or folders in one of these file explorers. An important note is that it is better to select the whole directory than selecting all the files individually, for the metric information is computed faster. The reason is that the files selected individually are analyzed separately. After the files are selected, the computation starts and computed results are displayed in the metrics views.



```php
<?php

include 'file2.php';
$a = $_POST['a'];

$class = new myclass;
$b = $class->get_result();

?>
```

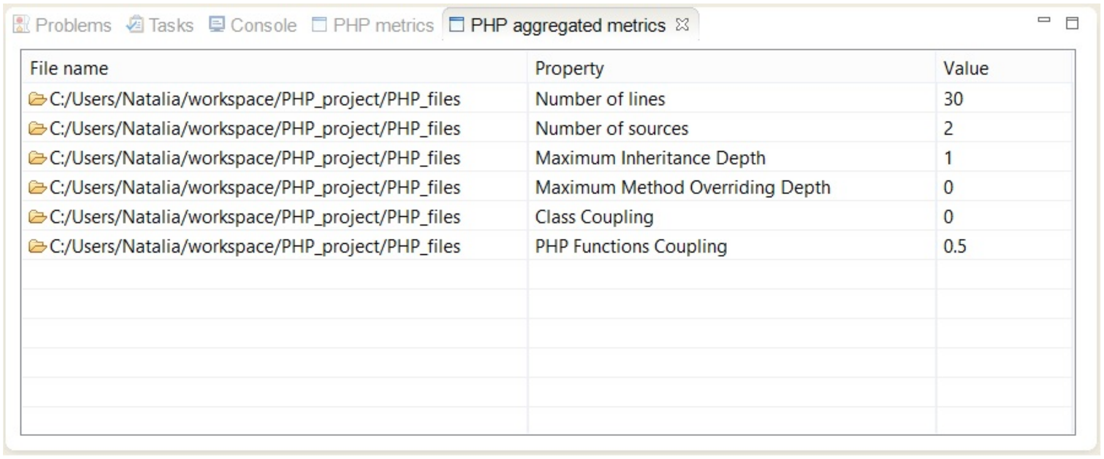
File name	Property	Value
C:/Users/Natalia/workspace/PHP_project/PHP_files	Number of lines	30
C:/Users/Natalia/workspace/PHP_project/PHP_files	Number of sources	2
C:/Users/Natalia/workspace/PHP_project/PHP_files	Maximum Inheritance Depth	1
C:/Users/Natalia/workspace/PHP_project/PHP_files	Maximum Method Overriding Depth	0
C:/Users/Natalia/workspace/PHP_project/PHP_files	Class Coupling	0
C:/Users/Natalia/workspace/PHP_project/PHP_files	PHP Functions Coupling	0.5

Figure 1.4: Location of the views

The analysis result contains following metrics:

- **Number of lines and Number of sources**
  These metrics provide a simple statistical information.
- **Maximum inheritance depth**
  This metric represents the longest chain of class inheritance.
- **Maximal depth of method overriding**
  The metric shows the number of implementations of the most frequently overridden method.
- **Class coupling**
  Class coupling represents an average number of user-defined classes that a single class uses. It is recommended to try to lower this number down, for highly coupled code contains a lot of dependencies and therefore is harder to reuse and maintain.
- **PHP functions coupling**
  This metric depicts an average number of user-defined functions that a single class calls. Once again a higher number equals worse application design.

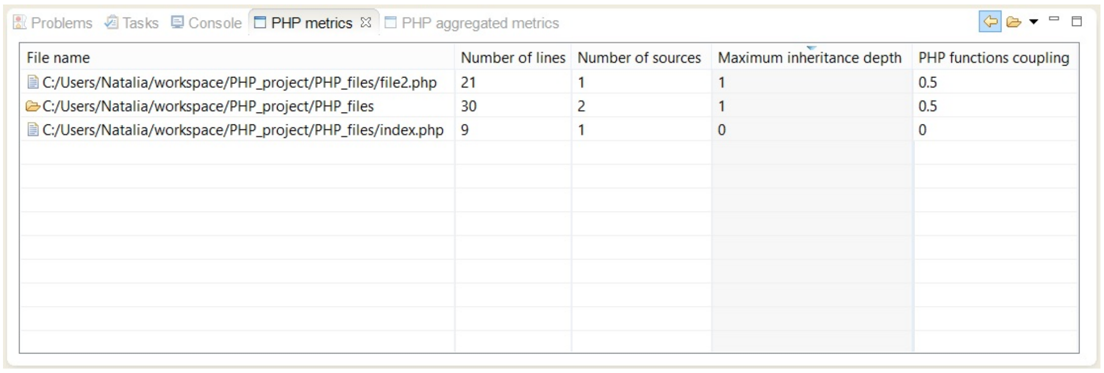
The PHP aggregated metrics view combines the metric information of all selected files and folders recursively. Combining means adding the numbers in case of Number of lines and Number of sources metric or choosing the maximum value otherwise.



File name	Property	Value
C:/Users/Natalia/workspace/PHP_project/PHP_files	Number of lines	30
C:/Users/Natalia/workspace/PHP_project/PHP_files	Number of sources	2
C:/Users/Natalia/workspace/PHP_project/PHP_files	Maximum Inheritance Depth	1
C:/Users/Natalia/workspace/PHP_project/PHP_files	Maximum Method Overriding Depth	0
C:/Users/Natalia/workspace/PHP_project/PHP_files	Class Coupling	0
C:/Users/Natalia/workspace/PHP_project/PHP_files	PHP Functions Coupling	0.5

Figure 1.5: PHP aggregated metrics view example

The PHP metrics view shows the metric information of each file individually. The metrics are shown in columns, which can be optionally hidden by right-clicking the table and deselecting the column. This view provides a possibility to choose whether to only show files or folders, or both of them. The user can also switch between analyzing the folders recursively and non-recursively. Both of these options can be found in a toolbar located in the view's upper right corner. It is also possible to sort the table rows according to a specific metric by clicking the column description.



File name	Number of lines	Number of sources	Maximum inheritance depth	PHP functions coupling
C:/Users/Natalia/workspace/PHP_project/PHP_files/file2.php	21	1	1	0.5
C:/Users/Natalia/workspace/PHP_project/PHP_files	30	2	1	0.5
C:/Users/Natalia/workspace/PHP_project/PHP_files/index.php	9	1	0	0

Figure 1.6: PHP metrics view example

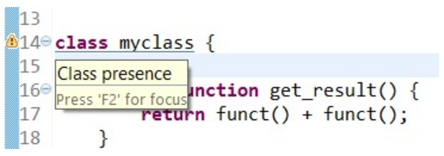
1.4.2 Constructs

The plug-ins provide a possibility to detect some special PHP elements, special constructs, variables or functions. These constructs are usually potentially dangerous or prevent the analyzer from completely analyzing the whole source code. The following constructs can be detected:

- **SQL**
  This indicates whether there is any MySQL function in the code. A complete list of these functions can be found in the PHP documentation [?].
- **Sessions**
  This construct comprises of session functions which allow preserving data across subsequent accesses.
- **Autoload**
  Autoload means use of function __autoload or spl_autoload_register. These functions allow to load a type that has not been declared yet.
- **Magic methods**
  This construct involves special methods that are usually called in a language construct. If a magic method is not declared, either an error is reported or a default operation is performed.
- **Class presence**
  Class presence checks for the class declarations in the source code.

- **Aliasing**
  This construct indicates an alias presence.
- **Inside function declaration**
  This involves function and types declared inside a subroutine. Locally declared function or type becomes global after the first declaration, however it cannot be declared multiple times. Developer should avoid inside function declarations.
- **Use of super global variable**
  Super global variables represent an application input, for example $_GET and $_POST variables. These arrays contain unpredictable content and should be always validated and sanitized.
- **Dynamic dereference**
  Dynamic dereference is potentially dangerous construct that allows usage of variable as a reference to another variable. Consider variable $a = "b", then the construct $$a is equivalent to $b. It is recommended not to use dynamic dereference, for the resulting variable might be indefinite.
- **Dynamic call**
  Dynamic call is similar to the dynamic dereference, only the dereferenced value is used as a name of method or class. This construct is even more dangerous that dynamic dereference and developer should avoid it.
- **Dynamic include**
  This construct allows to include a file (with additional source code) whose name is generated in run-time.
- **Eval**
  Eval is a very dangerous PHP construct, since it allows execution of arbitrary PHP code stored as a string value.
- **Passing by reference at call site**
  This means passing a references parameter to a function, so that the function is able to write to the original variable.

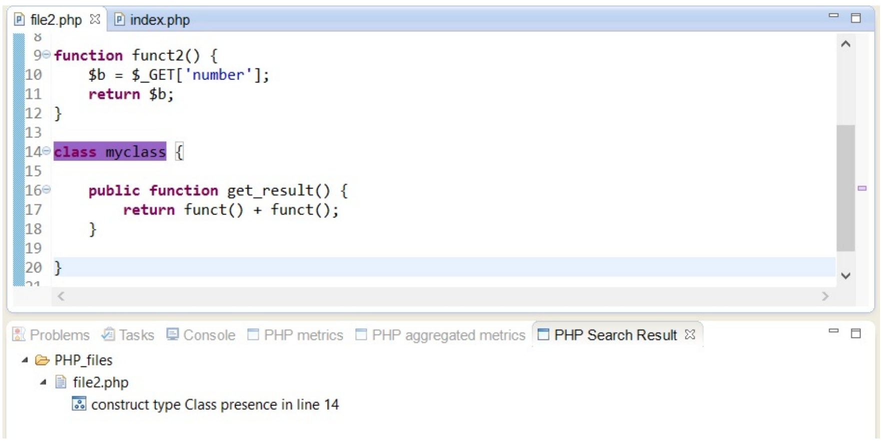
The constructs can be detected in two ways. The first way is to define the constructs that will be marked as warnings and displayed as the user is typing. Defining the set of constructs to be marked on-the-fly is described in section 1.3.



```
class myclass {
    Class presence
    Press 'F2' for focus nction get_result() {
        return funct() + funct();
    }
```

Figure 1.7: Class presence warning

The second way is to search for the construct occurrences manually. A search page was created for this purpose that can be found in the main toolbar: `Search -> Search -> PHP Construct Search`. The user can select which constructs to search for and whether to search in selected files (only PHP Explorer and Project Explorer selections are available) or in the whole workspace. If a folder is selected, it is searched recursively. After the search is processed, a PHP Search Result view is opened with the results. These results are clickable so that the developer is able to easily navigate to the desired construct occurrence. All the occurrences are also highlighted in the editors and the highlight colors can be defined as showed in section 1.3.



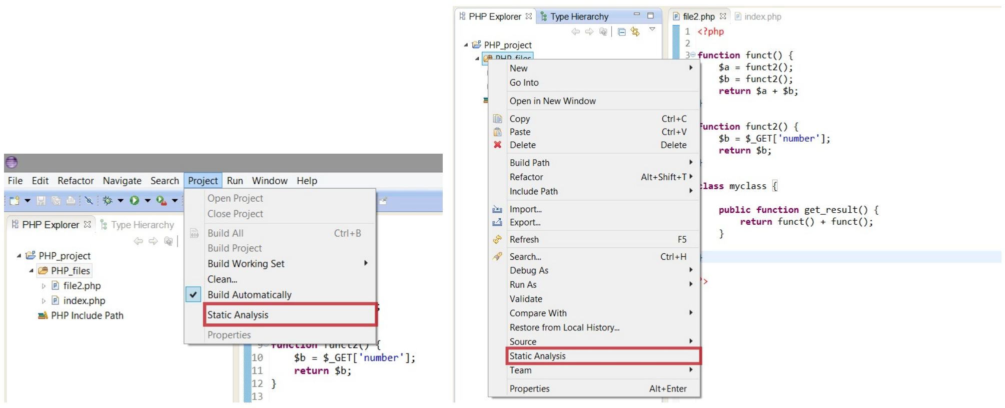
```
function funct2() {
    $b = $_GET['number'];
    return $b;
}

class myclass {

    public function get_result() {
        return funct() + funct();
    }

}
```

```
PHP_files
  file2.php
    construct type Class presence in line 14
```

Figure 1.8: Class presence search result

1.4.3 Static Analysis

Static analysis provides deeper information about the code. The analysis can be launched in two possible ways. First, by a menu button that can be found in the main toolbar `Project -> Static Analysis`. This option is only available when PHP editor is open, since it analyzes the open editor and other included files. Another option is to select the files to be analyzed in PHP Explorer or Project Explorer and right-click the selection. A pop-up window will show up containing a Static Analysis menu button.



```
<?php

function funct() {
    $a = funct2();
    $b = funct2();
    return $a + $b;


unction funct2() {
    $b = $_GET['number'];
    return $b;


lass myclass {

    public function get_result() {
        return funct() + funct();
    }


>
```

```
    $b = $_GET['number'];
    return $b;
}
```

Figure 1.9: Executing static analysis

When the analysis launches, a Static Analysis Overview view opens up. It allows to check whether the analysis is still running and also contains a button dedicated to terminating the analysis. This is particularly helpful in case the analysis is running for abnormally long time. After the analysis is done, this view displays the overall analysis information such as analysis time or number of processed program points.

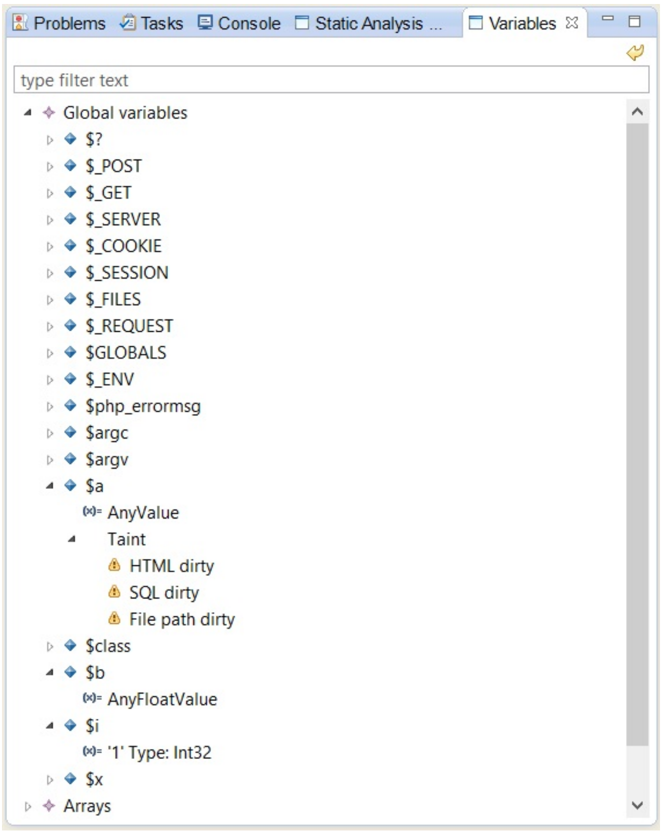
As a result of the analysis, two or three new views shows up. These views contain warnings, information about unreachable code and also provide an option to display the superset of possible values for each variable. All this information is context-sensitive which means that in case of extensions (e.g., bodies of functions that extends program points representing function calls), the information might be split into multiple parts. Each part corresponds to a specific context (e.g., place from which the function is called or more precisely call stack) and hence may contain different information. However, not every singe call has its own context, the contexts may be shared among multiple extensions. In any case, if a context-sensitive information is shown, there is a call stack provided representing the sequence, or possible sequences, of calls leading to the information.

Unreachable Code View

Unreachable Code view is only opened if there is any unreachable code in the analyzed files. This view shows each unreachable line in the analyzed files. If the line is a part of a code extension, a call stack that led to unreachable code is displayed too. Both unreachable lines and call stacks are clickable and allow a quick focus on the specified line. Additionally, in the view's top right corner there is a button dedicated to highlighting all the unreachable code.

Variable View

Another view that is automatically opened is a Variables view. This view only shows content when a caret is located in a line from analyzed source code. The view displays possible variable types and contents for each variable just before or after the line is processed. To set whether to show the information before or after line processing there is a toggle button located in the view's toolbar in the top right corner. If the line is a part of extension and there exist multiple contexts for the line, it shows the variable information for all of them, with clickable call stacks included.



```
Global variables
  $?
  $_POST
  $_GET
  $_SERVER
  $_COOKIE
  $_SESSION
  $_FILES
  $_REQUEST
  $GLOBALS
  $_ENV
  $php_errormsg
  $argc
  $argv
  $a
    AnyValue
    Taint
      HTML dirty
      SQL dirty
      File path dirty
  $class
  $b
    AnyFloatValue
  $i
    '1' Type: Int32
  $x
Arrays
```

Figure 1.10: Example of a Variables view

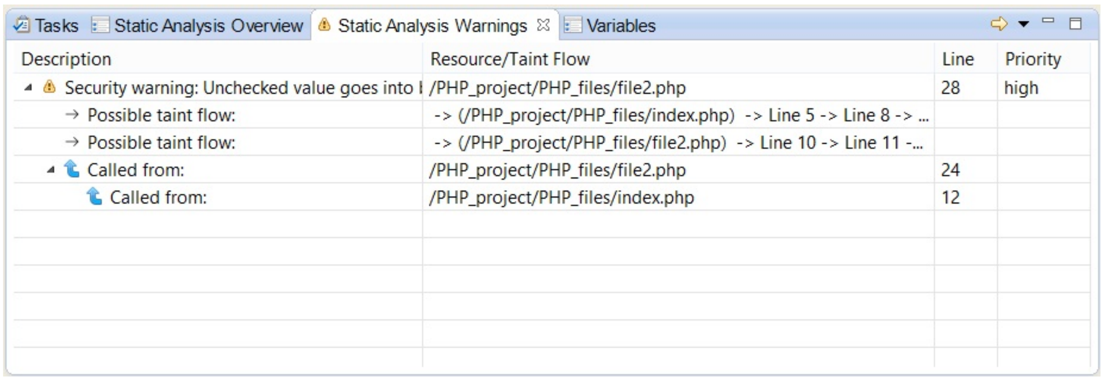
The actual information is divided into three parts:

1. **Global variables** that exist in the entire scope of the source code. For example _GET array.
2. **Local variables** which are only accessible in a specific context.
3. **Aliases**, which contain information about variable references.

Each part contains all the variables of given type from the current context. A variable contains a value and, in case it is tainted, also a list of taint flags. A value can be concrete (for example ''`True`' `Type: Boolean`') or it can be an interval of numbers, or an abstract value representing any value of a type (for example '`AnyIntegerValue`'. The only possible taint flags are `HTML dirty`, `SQL dirty` and `File path dirty`.

Warnings View

The last view with static analysis results is Warnings view, which contains all the warnings that were raised during the static analysis. All the warnings are clickable, so that it is possible to navigate to the code that caused the warning. In case of an extension, a call stack that led to a warning is showed too.



Description	Resource/Taint Flow	Line	Priority
Security warning: Unchecked value goes into I	/PHP_project/PHP_files/file2.php	28	high
→ Possible taint flow:	-> (/PHP_project/PHP_files/index.php) -> Line 5 -> Line 8 -> ...		
→ Possible taint flow:	-> (/PHP_project/PHP_files/file2.php) -> Line 10 -> Line 11 -...		
Called from:	/PHP_project/PHP_files/file2.php	24	
Called from:	/PHP_project/PHP_files/index.php	12	

Figure 1.11: Example of a Warnings view

There are two types of warnings—general warnings and security warnings. A general warning is for example '`Wrong number of arguments`'. Security warnings are related to the flows of tainted values, for example '`Unchecked value goes into browser`'. These warnings usually contain one or more possible taint flows that caused the warning. They shows flows of non-sanitized user input into a sink, e.g., a database or a browser.

There is a button in the view that provides four options to display the taint flows:

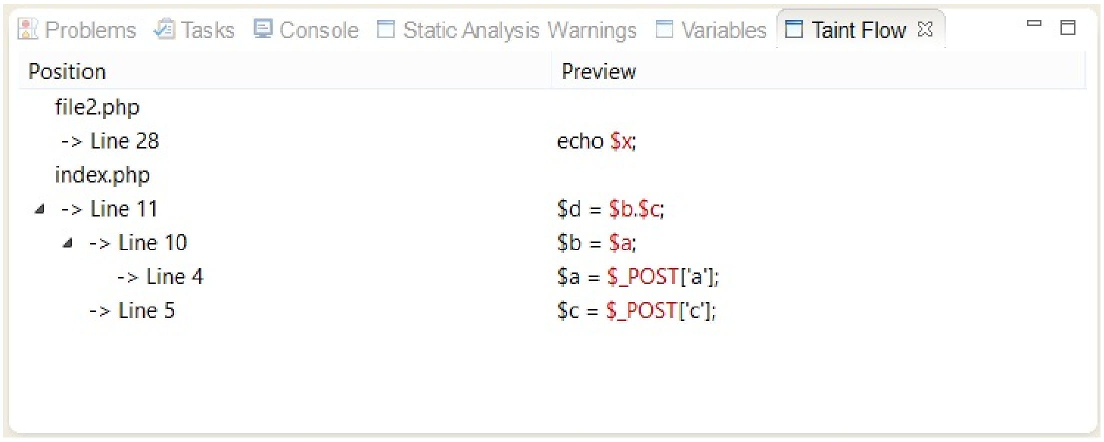
1. Separated and showed from the source to the sink.
2. Separated and showed from the sink to the source.
3. Merged by their source (showed from the source to the sink).
4. Merged by their sink (showed from the sink to the source).

In some cases it is possible that a displayed taint flow is unfeasible. Consider the following PHP code:

```php
<?php
$x = $_POST;
$a = 'a';
$b = $x;
if ($x) { $a = &$b; }
$b = 'str';
echo $a;
?>
```

The static analysis raises a warning for the last line, stating that unchecked value goes into browser. However, this warnings is unfeasible, for the flow would be sanitized by assigning `$b = 'str';`. Therefore it is necessary to always review the taint flows manually.

We made the review easier by making the taint flows in Warning view clickable. There is a new view called Taint Flow opened when a flow is double-clicked. Simultaneously, the flow is highlighted. To avoid confusion the highlight is only shown when the Taint Flow view is active. The Taint Flow view contains a list of the flow lines with a preview. In this preview, variables that carry the tainted information are red-coloured. Again, for easier navigation, these lines are clickable.



Position	Preview
file2.php	
-> Line 28	echo $x;
index.php	
-> Line 11	$d = $b.$c;
-> Line 10	$b = $a;
-> Line 4	$a = $_POST['a'];
-> Line 5	$c = $_POST['c'];

Figure 1.12: Example of a Taint Flow view

Warning

Even though the static analysis runs in the background so that it is possible to modify the files that are being analyzed, it is strongly recommended not to do so. If an analyzed file is modified during or after the analysis, the results might not be correct (for example the lines might be shifted, the taint flows might show absurd and false information etc.).