Weverca

Web verification tool for PHP

User documentation

1 Contents

2 Introduction

Weverca is a framework for static analysis which allows user to create their own static analyses. It also contains implementation of several analyses and thus can be used as a tool for error detection. It detects Cross-site scripting and SQL injection attacks. It also warns programmer if some unchecked input is used as a filename or in an eval construct. User also gets information about possible values of variables. Tools also generates warnings, which inform user about possible runtime errors.

Weverca was developed as a library, however it also contains a console application project.

2.1 System requirements:

- operating system Windows Vista SP2 or newer, windows server 2008 R2 SP1 or newer
- installed .NET framework 4.5 or newer

2.2 Supported PHP constructs

Tool currently supports PHP 5.1 with some constructs from PHP 5.3. Never versions are not supported because at the beginning of the development the syntax parser (Phalanger) used by Weverca supported only mentioned version of PHP.

Supported constructs:

- function and class declarations
- constants
- variable reading and writing (direct and indirect)
- static and non-static field reading and writing (direct and indirect)
- direct and indirect object creation
- all binary and unary expressions (arithmetical, logical, string expressions...)
- cycles (for, foreach, while, do while)
- conditions
- labels and goto statements
- exceptions
- function, methods and static method calls(direct and indirect)
- namespaces
- includes and evals
- array expressions

Not supported constructs:

- traits
- unset statement
- list expression
- shell statement

Traits are not supported, because Phalanger in version 3.0 was only able to parse it, but didn't provide complete information in syntax tree. Some other constructs will be supported in newer versions of analyzer.

2.3 Taint analysis

Important feature of Weverca is that it allows to detect security problems using static taint analysis. Static taint analysis can be used for detection, e.g., of SQL injection and cross-site scripting attacks.

Static taint analysis tracks sensitive *tainted* information through the application by starting at a *source* and reaching a *sink*. *Source* is the program point which reads user-input, session ids, cookies, or any other data that can be manipulated by a potential attacker, while *sink* is a program point that prints out the output, queries a database, etc. Data at a given program point are *tainted* if they can pass from a source to this program point. A tainted value is *sanitized* if it is processed by a sanitization routine (e.g., `htmlspecialchars in PHP)` to remove potential malicious parts of it. An execution of a PHP program is called *vulnerability* if data can pass from a source to a sink without being sanitized.

Cross-site scripting (XSS) is a security attack involving stealing credentials (e.g., a cookie), which is accessible just by the code of the corresponding page by injection of malicious Javascript [1]. Similarly, a SQL injection (SQLI) aims at execution of custom queries on a (SQL) database via injecting a malicious SQL commands. Both XSS and SQLI can be detected by means of taint analysis.

Programmer can prevent this types of attacks by sanitizing (using PHP functions like `htmlspecalchars` or `mysql_escape_string`) input variables before using them in database queries or writing them back to the browser. If the programmer is expecting a number from user input he or she can sanitize this values by using `intval` function or its equivalent.

3 User documentation

The tool provides various types of information about PHP code. The desired type of output is specified via command-line arguments described in following chapter.

3.1 Metrics

Usage: `weverca.exe -metrics filename [filenames ...]`

Program performs analysis of source code and prints values of all compiled metrics. Weverca has pre-implemented a bunch of some standard and PHP-specific metrics. The output of analysis is a list of results, one for every metric. They are divided into three categories: Indicator, rating and quantity.

Indicators say whether a property has appeared in the code or not. It is usually used for detecting some elements of language (e.g. classes), special construction (e.g. declaration of function inside another function) or special variables or functions (MySQL functions or super globals variables). Rating metrics represents a "score", evaluation of some more complex property. Only class and function cyclomatic complexity is currently implemented. Quantity in metric means a number of some property. The property that can be counted (e.g. number of lines) or it reflects an extreme (maximum inheritance depth).

We can analyze more files at once. In this case, results of metrics are merged. Every metric has its own merging policy. However, there is implemented default behavior for every category:

- Indicator metric is set if at least one piece of code indicates the property.
- Rating values are averaged.
- Values of the quantity metrics are added.

3.1.1 Class presence

A model with objects is generally harder to analyze than other one with only scalar values. Class presence metric determines whether there is at least one class declared in the given code.

3.1.2 MySQL or other SQL functions usage

This metric checks if there is any of MySQL functions used in the code. Full list of the function is located in the PHP documentation (http://www.php.net/manual/en/book.mysql.php).

3.1.3 Dereference with double $

Dynamic dereference is one of the potentially dangerous constructs, which PHP allows to use. For example consider variable `$a = "b"` and construct `$$b`. In this case the last construct is equivalent to `$b`. The content of the variable `$a` might not be so clear in a real code. Therefore it is not recommendable to use construct like the one above. This metric checks if there is dynamic dereference of a variable present in the code.

3.1.4 Maximal depth of method overriding

The metric finds a method that is overridden most often and returns number its new implementations.

3.1.5 Maximum inheritance depth

The metric finds all the longest chain of inheritance from the most derived class to the class without ancestor and returns its length.

3.1.6 Dynamic calls and object creations

Dynamic call is similar to the dynamic dereference, but in this case the dereferenced value is not used as a name of a variable, but as a name of a method or class. For example consider an instance $a of class A and a variable `$var = "method"`. Construct `$a.$var()` is in this case equivalent to `a.method()`. Using of constructions like this one might be even more dangerous than using dynamic dereference. The metric checks if there are such constructions.

3.1.7 Class alias

In PHP there is a possibility to create an alias of a class. After applying this process, the class will have multiple names – one original and as many aliases as needed. Class alias is created by calling *class_alias* (http://www.php.net/manual/en/function.class-alias.php). The metric reports whether a creation of class alias is used.

3.1.8 Magic methods

Some methods in PHP classes has special functionality. They are usually not called directly, but in a language construct. They are called "Magic methods". If the method is not declared, run-time either reports an error (e.g. `__toString`) or performs a default operation (e.g. `__construct`). The metric checks presence of these methods.

3.1.9 Dynamic inclusion

Include statement is a mechanism to embed a file with other part of source code. Name of included file can be generated in run-time which in fact prevents from performing an adequate analysis. We must identify, if parameter of the statement can be evaluated in compile-time, so if the expression contains only literals and concatenation. If not, it must be evaluated dynamically. The metric determine if there is a dynamic inclusion.

3.1.10 Number of lines and source files

These metrics are only simple statistics.

3.1.11 Superglobals

Super global variable are built-in variables always defined in every script. They represent kind of input from some different sources. These are `$_GET` and `$_POST` for instance, coming from HTTP protocol, or `$_ENV` array with environment variables. These arrays can possibly contain any values and they are predictable at all. It can treasure sensitive or, vice versa, dangerous data, both are candidates for sanitation. The metric checks whether there are these variables used.

3.1.12 Function/type declaration inside function body

Functions and types declared inside subroutine body are parsed differently than global declaration. Declaration in global scope are accessible in the entire program, even if they are declared at the end of source code, whereas locally defined types and functions must be "called". When they are declared at the first time, they became global as any other type or function. However, if they are defined multiple times, it is fatal error. Generally, it is unwise to make local declarations. The metric detects whether there are such function/type declarations.

3.1.13 Duck typing

Duck typing is a type system where object's methods and properties determine the right behavior. It is opposite to system that determine type of object by inheritance. PHP has the first system, because the type of variable is not known in compile type, nor its interface, nor even if it is an object. The metric finds whether there is an access to object by a member.

3.1.14 Passing variable by reference at call side

When a subroutine is called, all parameters are copied. The function itself does not access the original variable, but its copy. Referenced parameters behave differently. The function parameter is the same variable as the variable at call side. The analysis must know to recognize reference in parameter, otherwise writing into this variable will be incorrect. The metric finds out whether there is subroutine that passes at least one parameter by reference.

3.1.15 Autoload

In order to avoid the need to add tons of includes into source code, PHP invented mechanism that tries to load currently used type that has not been declared yet. The function that is responsible for finding of the proper type is called `__autoload`. In later PHP versions, function `spl_autoload_register` has appeared. It provides a possibility to register arbitrary subroutine as new `__autoload` function. The metric finds out whether `__autoload` or `spl_autoload_register` are used.

3.1.16 Class coupling

Class coupling is a measure of how many user-defined classes a single class uses, for all classes on average. The higher quotient is, the more design suffers. Highly coupled source code indicates problems to reuse and maintain because of the many dependencies between each other.

The metric finds all couplings between classes, i.e. for every class, all occurrences of other classes inside it. The rating is ratio between sum of all occurrences and number of classes. This is static information acquired from static references that can appear when there is object creation or static method call.

3.1.17 Functions coupling

This is similar to class coupling. It is measure of how many user-defined functions a single function calls on average. Again the higher quotient is, the worse the design is. The metric finds all couplings between functions, i.e. for every function, all other function calls (not recursion) inside its body. The rating is ratio between sum of all function calls and number of functions. Methods are taken as part of classes.

3.1.18 Eval

`eval` function is technique to execute a code in string value. This language construct is very dangerous because it allows execution of arbitrary PHP code. String in parameter can contain such a code that can absolutely change all data of analysis. Constant string can be evaluated and converted into real code, but evaluation of dynamically created strings can be very difficult or almost impossible. The metric detects use of `eval`.

3.1.19 Session functions

PHP Sessions are the way to preserve data across subsequent accesses. It solves the problem that HTTP protocol is stateless. The advantage for PHP is obvious, but analysis cannot use it. We cannot assume anything about values in `$_SESSION` array and session functions are appropriate candidate for performing a taint analysis. The metric checks whether there is a call of a session function.

3.1.20 References

In PHP there is a possibility to create an alias of a variable using reference syntax. For example consider a variable `$a` of any value. Using construct like `$b = &$a` will create an alias which can be used in the same way as variable `$a`. This metric check if there is such construct.

3.1.21 Example

Input: `weverca.exe -metrics test.php`
Output:
```
File path: test.php

Indicator Metrics

__autoload redeclaration presence or spl_autoload_register call: NO
Magic function presence: NO
```

```
Class construct presence: YES
Include based on dynamic variable presence: NO
Alias presence: NO
Session usage: NO
Declaration of class/function inside another function presence: NO
Super global variable usage: NO
Eval function usage: NO
Dynamic call presence: NO
Dynamic dereference presence: NO
Duck Typing presence: NO
Passing variable by reference at call site: NO
My SQL functions presence: NO
Class alias construction presence: NO

Rating Metrics

Class coupling: 0
PHP standard functions coupling: 0

Quantity Metrics

Maximum inheritance depth: 1
Number of lines: 7
Number of sources: 1
Maximum method overriding depth: 0
```

3.2 Static analysis

Usage: `weverca.exe -sa [-mm CopyMM|VrMM] filename [filename...]`

The main goal of Weverca is to perform static analysis of PHP. It is executed over list of files containing PHP code and generates comprehensive output with results of analysis if analyzer reaches endpoint, the last node of control flow graph. Part of the output is memory dump at the end of the program, listing of possible warnings and, last but not least, results of taint analysis. If analysis detects that program never reach the end, it cannot create any output and just prints "Point not reached".

Static analyzer computes snapshot of memory in every point of code. To store all information, it needs proper memory model. Weverca has implemented two memory models that can be passed with `–mm` option: *Virtual Reference* (VrMM) and *Copy memory model* (CopyMM). The Virtual Reference is the simpler one and it is useful for testing of static analysis framework. On the other hand, Copy memory model is more precise but slower. Virtual Reference memory model is the default one.

3.2.1 Values in memory model

Unlike dynamic analysis, static analysis does not execute the program, but it creates its own context in every point of code. Since analysis traverse every possible path in source code (every path in control flow graph), every assigned place in memory (called memory entry) can contain a set of possible values. PHP is not typed, so variables may even contain values of different types. It is impractical to store large number of values, therefore framework can internally work with abstract values, which represent abstraction of a set of values. Weverca has two types of abstract values: any values (e.g., AnyValue, AnyIntergerValue and AnyStrngValue) and intervals. We enumerate all possible types of values:

- Concrete values
    - Boolean values (e.g. `''True' Type: Boolean'`)
    - Integers (e.g. `''-123' Type: Int32'`)
    - Floating-point numbers (e.g. `''3.141529' Type: Double'`)
    - Strings (e.g. `'apples and pears'`)
    - Objects (e.g. `'ObjectValue UID: 1'`)
    - Arrays (e.g. `'AssociativeArray UID: 2'`)
    - Resources (`'ResourceValue'`)
    - Undefined value that is called NULL in PHP (`'UndefinedValue'`)
- Intervals of numbers
    - Integer intervals (e.g. `''(-16,16)', Type: Int32'`)
    - Floating-point intervals (e.g. `''(24.74,26.18)', Type: Double'`)
- Abstract values representing any value of a type
    - Universal abstract value (`'AnyValue'`)
    - Abstract boolean value (`'AnyBooleanValue'`)
    - Abstract integer (`'AnyIntegerValue'`)
    - Abstract floating-point number (`'AnyFloatValue'`)
    - Abstract string (`'AnyStringValue'`)
    - Abstract objects (`'AnyObjectValue'`)
    - Abstract arrays (`'AnyArrayValue'`)
    - Abstract resource (`'AnyResourceValue'`)

Every value can carry flags for later taint analysis. Every instance of object or array has unique identification number. Properties of object or elements of array is associated with this ID.

3.2.2 Format of memory model output

The memory model dumps all data stored in the memory at the endpoint of the analysis. Some memory places have its names and we call them variables. However, they are not always such variables as we know from PHP runtime. Some of them are control variables that Weverca uses for its own purposes. Data not accessible by name are called meta information. Usually, these are properties of objects or elements of arrays.

Both memory models in Weverca have its own outputs. It is generally almost the same, but the format can be slightly different and results can vary, because they approach memory differently. The memory dump is divided into sections labeled with name in `===NAME===` format. Every line in section denotes one memory entry. It is characterized by identification, that can be name in case of variables, or a special string reflecting the content of memory the location. Then follows list of all possible values that can be placed in memory entry at the end of the analysis.

We can divide variables to global and local in PHP. The global variables exist in the entire scope of the code. They are listed in `===GLOBALS===` section of output and are marked as `Global` for VrMM. For instance, line "`_GET|Global: {'AnyArrayValue'}`" for VrMM and "`$_GET: { Values: (AnyArrayValue) }`" for CopyMM says that global variable `_GET` has only one value, abstract representation of array.

If we want to access global variables inside function body, the variable must be previously fetched by `global` keyword. There exist arrays that are accessible inside subroutines regardless of whether they were fetched or not. They are called Superglobal variables and their names start with underscore (e.g. `_GET`, `_POST` or `_SESSION`). The other variables are either passed into program by PHP engine (e.g. `argc`, `argc` or `php_errormsg`) or initialized by programmer.

Global control variables are elegant way to store data of analysis for every program point. They are declared in other namespace than regular variables, so they cannot influence the analysis. They can be identified as variables with dot at the beginning of name. Here is the list of them:

- `.staticVariables` - Array of statically defined properties of classes. Every class initializes index of its name. The element is an array that contains all statically declared properties of the given class.
- `.analysisWarning` - Analysis warnings identified during static analysis
- `.analysisSecurityWarning` - Security warnings identified during taint analysis
- `.catchBlocks` - Stack of visited try statements with associated catch blocks. It is used for handling exceptions.
- `.constants` - Constants declared in source code. They can have more possible values if they are declared conditionally, in a subroutine or condition branch.
- `.staticVariableSink` - Special field for writing and reading non-existing static variables
- `.evalDepth` - Integer value that denotes recursion of `eval` calls, i.e. depth of `eval` call inside itself.
- `.includedFiles` - List of included files in `include` and `include_once` statement.
- `.class(CLASS_NAME)->constant(PROPERTY_NAME)` - All possible values of class constant `CLASS_NAME::PROPERTY_NAME`. There can be more of these control variables.

Properties of objects and elements of arrays are memory entries like any other, but they must be accessed both by name and identifier of the compound type. VrMM prints all property entries into `===META===` section, every property occupies one line. For instance, property "`cost`" with value `1000.0` is represented by "`$obj123[cost]|Meta: {''1000.0' Type: Double'}`" line, where number `123` is identification of the object. CopyMM has its own `===FIELDS===` sections for them and the previously mentioned example is printed as "`123->cost: { Values: ('1000.0' Type: Double) }`".

Array elements in VrMM output are printed into `===META===` section, the same as properties. Even the output is similar, for instance, element in index "`5`" with value `0.47` has the form "`$arr123[5]|Meta: {''1000.0' Type: Double'}`". CopyMM has more sophisticated output. It has its own sections `===ARRAYS===` for elements and if array is stored in a variable, elements are printed the same way as in the PHP. For instance, if the array from example is stored in variable "`vector`", the line looks like this: "`$vector[5]: { Values: ('0.47' Type: Double) }`". Indices of other temporary arrays are just prefixed by "`TEMP--`" string. The objects cannot be printed like that, because more variables can hold reference to them.

There are other meta information. VrMM stores references of defined types and functions. A type with name `TYPE_NAME` is identified as "`$type: TYPE_NAME|Meta`" and a function with name `FUNCTION_NAME` as "`$function-FUNCTION_NAME|Meta`".

On the other hand, CopyMM dumps all information about PHP variable references in section `===ALIASES===`. For every variable, there is a line with all references to the same memory place, if there are any. There are *MUST references*, which certainly point to the same memory entry, and *MAY references*, which may point to another place or nowhere.

In Copy memory model, there exists special memory entry known as *unknown field*. When the identification of property or index cannot be resolved only into concrete values (even property of object may begin with an expression in PHP), it is not clear where the value should be stored. The unknown field corresponds to any potential place in compound value. CopyMM represents such property resp. index by question mark (e.g. "`$vector[?]: { Values: ('0.47' Type: Double) }`").

3.2.3 Possible analysis warnings

When static analysis is running, evaluating expressions, and computing fix-point, it has a lot of information to detect various problems in a analyzed source code. It reports them as warnings with position in source code. Note that most of these warnings would be reported also by the PHP engine at runtime.

In this subchapter, we list all possible analysis warnings. Exceptions and fatal errors change flow of the program, but are not reported, because PHP runtime is checking them by its own.

The following warnings occur during class and interface declaration. PHP interpreter ends with fatal error in this cases:

- Class X doesn't exist
- Class not found when creating new object
- Cannot redeclare class/interface X
- Cannot extend final class X
- Class X doesn't implement method Y
- Interface X not found
- Cannot redeclare non static x::$a with static y::$b
- Cannot redeclare static x::$a with non static x::$b
- Cannot redeclare non static method with static X

- Cannot redeclare static method with non static X
- Cannot redeclare final method X
- Cannot override interface constant x
- Cannot redeclare constant x
- Cannot redeclare field x
- Cannot redeclare method x
- Non abstract class cannot contain abstract method x
- Can't override function x, with abstract function
- Can't inherit abstract function x because arguments doesn't match
- Interface cannot contain fields
- Interface method x must be public
- Interface method x cannot be final
- Interface method x cannot have body
- Abstract method x cannot have body
- Non abstract method x must have body

Function or method call warnings:

- Wrong number of arguments
- Wrong type in argument No. 1 in function strstr, expecting string - occurs
- Function x doesn't exists
- Function x already exists
- Cannot call method on non object
- Cannot call function without body
- Calling inaccessible method

Warnings, that occurs during eval or include analysis:

- Couldn't resolve all possible evals
- Parser error in eval
- Control flow graph creation error in eval
- Couldn't resolve all possible calls
- Couldn't resolve all possible includes
- Eval cannot be called in "eval recursion" more than 3 times
- The file x.php to be included and not found

Warning that occurs, when working with object fields and constants:

- Trying to get element of scalar value
- Static variable X::a wasn't declared
- Cannot access static variable on non-object
- Accessing inaccessible field
- Cannot access class constant on non object
- Constant X::a does not exist
- Cannot use operator -> on variable other than object

Other warnings:

- Division by zero

- Object cannot be converted to integer by arithmetic operation
- Trying to get property of non-object
- Only objects derived from Exception can be thrown
- Cannot use keyword self when it is not in class
- Cannot use keyword parent when it is not in class
- Cannot use keyword parent, class has no parent
- Possible use of undefined constant
- Cannot instantiate abstract class X
- Cannot instantiate interface X
- Cannot use operator [] on variable other than string or array
- Cannot index string with negative numbers

Security warnings

The main objective of Weverca is to identify potential security bugs. As already described in the introduction, the program identifies three types of potential security violations. Three warnings correspond to the three types of bugs:

- "`Unchecked value goes into browser`" - It occurs when writing data from user input into browser. Data may contain elements of HTML or JavaScript code that can change the page layout at the best case, its behavior in the worst case. Usually, `htmlentities` and `htmlspecialchars` functions can sanitize the input.
- "`File name has to be checked before open`" - It occurs when trying to include or open a file with name containing part of user input. It may lead to opening a private or system file.
- "`Unchecked value goes into database`" - It occurs when sending database query that contains part of unchecked user input, e.g. into `mysql_query` function. There is a possibility of a SQL injection. The example of sanitization function is `mysql_escape_string`.

3.2.4 Function hints

Sometimes programmers use their own functions to sanitize values from user input. For example, they use series of `str_replace` instead of `htmlspecialchars` function. Analyzer cannot detect if the `str_replace` makes the required sanitation. The example below describes how a PHP programmer can tell the analysis framework that the result of the function is sanitized.

Example:

```
/**
 * @wev-hint returnvalue remove all
 */
function fn($str)
{
      $str = str_replace("<", "&lt;", $str);
      return $str;
}
echo fn($_POST["A"]);
```

This example does not produce security warning, because analyzer uses the information provided by a programmer in PHP document comments.

Hints have following syntax:

```
 * @wev-hint sanitize [flagType]
```

Flag types:

- HTMLDirty - value cannot go into browser
- SQLDirty - value cannot be send into database
- FilePathDirty - value cannot be used as file name
- all - all of above

3.3 Examples of Usage

Example 1

Source code:

```
<?php
/* 2 */ $input = $_GET['anywhere'];
/* 3 */ $a[1][1] = 'apple';
/* 4 */ $a[1][2] = 'pear';
/* 5 */ $a[$input][1] = $input;
/* 6 */ echo $a[1][2]; // not XSS
/* 7 */ echo $a[1][1]; // XSS
?>
```

Command line input: `weverca.exe -sa -mm CopyMM example1.php`

Output:

```
Using Copy memory model

File path: example1.php

PROGRAM POINT: End point
      ===GLOBALS===
      $?: { Values: (UndefinedValue) }
      $_POST: { Values: (AnyArrayValue) }
      $_GET: { Values: (AnyArrayValue) }
      $_SERVER: { Values: (AnyArrayValue) }
      $_COOKIE: { Values: (AnyArrayValue) }
      $_SESSION: { Values: (AnyArrayValue) }
      $_FILES: { Values: (AnyArrayValue) }
      $_REQUEST: { Values: (AnyArrayValue) }
      $GLOBALS: { Values: (AnyArrayValue) }
      $_ENV: { Values: (AnyArrayValue) }
      $php_errormsg: { Values: (AnyStringValue) }
      $argc: { Values: (AnyIntegerValue) }
      $argv: { Values: (AnyArrayValue) }
      $input: { Values: (AnyValue) }
      $a: { Values: (AssociativeArray UID: 2235) }

      ===GLOBAL CONTROLS===

      ===LOCAL CONTROLS===
      CTRL--$?: { Values: (UndefinedValue) }
      CTRL--$.staticVariables: { Values: (AssociativeArray UID: 133) }
      CTRL--$.analysisWarning: { Values: (UndefinedValue) }
      CTRL--$.analysisSecurityWarning: { Values: (Warning at line 7
char 9: Unchecked value goes into browser) }
      CTRL--$.catchBlocks: { Values:
(Weverca.Analysis.FlowResolver.TryBlockStack) }
      CTRL--$.constants: { Values: (AssociativeArray UID: 174) }
      CTRL--$.staticVariableSink: { Values: (UndefinedValue) }
      CTRL--$.evalDepth: { Values: ('0' Type: Int32) }


      ===ARRAYS===
      TEMP--$24[?]: { Values: (UndefinedValue) }

      TEMP--$31[?]: { Values: (UndefinedValue) }

      CTRL--$.staticVariables[?]: { Values: (UndefinedValue) }

      CTRL--$.constants[?]: { Values: (UndefinedValue) }

      $a[?]: { Values: (UndefinedValue),(AssociativeArray UID: 2323) }
      $a[1]: { Values: (AssociativeArray UID: 2244) }

      $a[1][?]: { Values: (UndefinedValue) }
      $a[1][1]: { Values: (apple),(AnyValue) }
      $a[1][2]: { Values: (pear) }

      $a[?][?]: { Values: (UndefinedValue) }
      $a[?][1]: { Values: (UndefinedValue),(AnyValue) }
```

```
        ===FIELDS===

        ===ALIASES===
        $_POST: {  MUST: $_POST | MAY }
        $_GET: {  MUST: $_GET | MAY }
        $_SERVER: {  MUST: $_SERVER | MAY }
        $_COOKIE: {  MUST: $_COOKIE | MAY }
        $_SESSION: {  MUST: $_SESSION | MAY }
        $_FILES: {  MUST: $_FILES | MAY }
        $_REQUEST: {  MUST: $_REQUEST | MAY }
        $GLOBALS: {  MUST: $GLOBALS | MAY }
        $_ENV: {  MUST: $_ENV | MAY }
        $php_errormsg: {  MUST: $php_errormsg | MAY }
        $argc: {  MUST: $argc | MAY }
        $argv: {  MUST: $argv | MAY }


Analysis completed in: 1283ms


Analysis warnings:

No warnings

Security warnings:


File: example1.php

Warning at line 7 char 9: Unchecked value goes into browser
```

Example 1 demonstrates XSS vulnerability when accessing statically unknown field. Program reads superglobal variable `$_GET`. It contains an array of variables passed via URL parameters and it is impossible to deduce values of elements with static analysis. There is a risk of any kind of violation when the script uses these values. Therefore all its elements are tainted with all security flags by default, including XSS flag.

Program creates an two-dimensional array in variable `$a`. It initializes index 1 in the array to another array and then sets indices 1 and 2 of that second array to constants 'apple' and 'pear' respectively. Then it updates `$a[$input][1]` memory place, where `$input` variable has statically unknown value, with the variable `$input` itself that contains tainted value with XSS flag. Statement "`$a[$input][1] = $input;`" is equivalent to "`$a[$input] = array(1 => $input);`" and since the new array can be written to any index of `$a` array, Copy memory model appends it to list of possible values of every index in the `$a` array.

As it can be seen in the output, at first, an array is created and appended into `$a[?]` unknown index. There must be NULL value too, because it is the value of uninitialized memory. Then tainted value is appended to possible values in `$a[?][1]`, but also in `$a[1][1]`, because this is the true meaning of unknown field, value can be possibly stored to any indexed place. Note that if we would read `$a[2]`, we would get actual value of unknown field.

The security warning is reported at line 7, because it may print tainted value of `$a[1][1]`. These warnings are stored in control variable "`.analysisSecurityWarning`". Unknown field works even for global variables, e.g. if we have assignment "`$$_GET['anywhere'] = 'orange';`". The value is appended to memory entry of variable printed to the output as "`$?`".

Example 2

Source code:

```php
<?php
/*  2 */ function fnA($obj) {
/*  3 */    global $functionMessage;
/*  4 */    $functionMessage = "fnA was called.";
/*  5 */ }
/*  6 */
/*  7 */ function fnB($obj){
/*  8 */    global $functionMessage;
/*  9 */    $functionMessage = "fn2 was called.";
/* 10 */ }
/* 11 */
/* 12 */ if ($_POST['anyvalue']) {
/* 13 */    class Window {
/* 14 */        protected $width = 32;
/* 15 */        function __construct() {}
/* 16 */    };
/* 17 */
/* 18 */    $indirectFunction = 'fnA';
/* 19 */ } else {
/* 20 */    class Window {
/* 21 */        protected $width = 48;
/* 22 */        function __construct() {}
/* 23 */    };
/* 24 */
/* 25 */    $indirectFunction = 'fnB';
/* 26 */ }
/* 27 */
/* 28 */ $window = new Window();
/* 29 */ $indirectFunction($window);
?>
```

Command line input: `weverca.exe -sa -mm VrMM example2.php`
Output:
```
Using Virtual reference memory model

File path: example2.php

PROGRAM POINT: End point
      ===GLOBALS===
      _POST|Global: {'AnyArrayValue'}
      _GET|Global: {'AnyArrayValue'}
      _SERVER|Global: {'AnyArrayValue'}
      _COOKIE|Global: {'AnyArrayValue'}
      _SESSION|Global: {'AnyArrayValue'}
      _FILES|Global: {'AnyArrayValue'}
      _REQUEST|Global: {'AnyArrayValue'}
      GLOBALS|Global: {'AnyArrayValue'}
      _ENV|Global: {'AnyArrayValue'}
      php_errormsg|Global: {'AnyStringValue'}
      argc|Global: {'AnyIntegerValue'}
      argv|Global: {'AnyArrayValue'}
      indirectFunction|Global: {'fnA', 'fnB'}
      window|Global: {'ObjectValue UID: 2027', 'ObjectValue UID: 2029'}
      functionMessage|Global: {'fnA was called.', 'fn2 was called.'}

      ===GLOBAL CONTROLS===
      .staticVariables|GlobalControl: {'AssociativeArray UID: 22'}
      .analysisWarning|GlobalControl: {'UndefinedValue'}
      .analysisSecurityWarning|GlobalControl: {'UndefinedValue'}
      .catchBlocks|GlobalControl:
{'Weverca.Analysis.FlowResolver.TryBlockStack'}
      .constants|GlobalControl: {'AssociativeArray UID: 27'}
      .staticVariableSink|GlobalControl: {'UndefinedValue'}
      .evalDepth|GlobalControl: {''0' Type: Int32'}

      ===LOCAL CONTROLS===



      ===META===
      $arr21#info|Meta: }
      $arr22#info|Meta: }
      $arr26#info|Meta: }
      $arr27#info|Meta: }
      $function-fnA|Meta: {'SourceFunctionValue fnA'}
      $function-fnB|Meta: {'SourceFunctionValue fnB'}
      $arr2022#info|Meta: }
      $arr2023#info|Meta: }
      $arr22[window]|Meta: {'AssociativeArray UID: 2023',
'AssociativeArray UID: 2016'}
      $type: window|Meta: {'TypeValue type: Window', 'TypeValue type:
Window'}
      $arr2015#info|Meta: }
      $arr2016#info|Meta: }
      $obj2027#info|Meta: {'TypeValue type: Window'}
      $obj2027->width|Meta: {''32' Type: Int32'}
      $obj2029#info|Meta: {'TypeValue type: Window'}
      $obj2029->width|Meta: {''48' Type: Int32'}


Analysis completed in: 883ms


Analysis warnings:

No warnings

Security warnings:

No warnings
```

Example 3 shows how the memory model looks at the end of the computation, if the code has an indeterminate condition. All 3 global variables defined in the program may have values from both branches of the condition. There are both `'fnA'` and `'fnB'` names of functions in `$indirectFunction` variable. Since `$indirectFunction` behaves like variable function, analysis must call both appropriate functions. Therefore, as a proof, variable `$functionMessage` has two values, each of them set in one function.

There are also definitions of two classes with the same names, each in one condition branch. PHP allows definition of a type of the given name only once, another definition causes fatal error. However, the analysis must be able to allow multiple definitions, unless it knows which to use. Variable `$window` is initialized to new objects. They are instances of different classes, but with the same name. We can see that one object with ID 2027 has property `width` with the value 32 and other one with ID 2029 has the same property with the value 48.

Example 3

Source code:

```
<?
/* 2 */ while (TRUE) {}
?>
```

Command line input: `weverca.exe -sa -mm CopyMM example3.php`
Output:

```
Using Virtual reference memory model

File path: example3.php

End point was not reached

Analysis completed in: 735ms


Analysis warnings:

No warnings

Security warnings:

No warnings
```

Example 3 demonstrates a situation where the program obviously never ends. Analysis can detect that the condition is always true and it does not deal with the other branch which is the rest of the program. Therefore we do not know anything about the last program point.

Example 4

Source code:

```
<?php
/*  2 */ $fruits[0] = 'apple';
/*  3 */ $fruits[1] = 'pear';
/*  4 */ $fruits[2] = 'orange';
/*  5 */
/*  6 */ $result = 'unknown';
/*  7 */ foreach ($fruits as &$value) {
/*  8 */     if ($value == $_GET['fruit']) {
/*  9 */         $result = $value;
/* 10 */     }
/* 11 */ }
/* 12 */
/* 13 */ $isOrange = FALSE;
/* 14 */ foreach ($fruits as &$value) {
/* 15 */     if ($value == 'orange') {
/* 16 */         $isOrange = TRUE;
/* 17 */         break;
/* 18 */     }
/* 19 */ }
?>
```

Command line input: `weverca.exe -sa -mm CopyMM example4.php`
Output:

```
Using Copy memory model

File path: example4.php

PROGRAM POINT: End point
      ===GLOBALS===
      $?: { Values: (UndefinedValue) }
      $_POST: { Values: (AnyArrayValue) }
      $_GET: { Values: (AnyArrayValue) }
      $_SERVER: { Values: (AnyArrayValue) }
      $_COOKIE: { Values: (AnyArrayValue) }
      $_SESSION: { Values: (AnyArrayValue) }
      $_FILES: { Values: (AnyArrayValue) }
      $_REQUEST: { Values: (AnyArrayValue) }
      $GLOBALS: { Values: (AnyArrayValue) }
      $_ENV: { Values: (AnyArrayValue) }
      $php_errormsg: { Values: (AnyStringValue) }
      $argc: { Values: (AnyIntegerValue) }
      $argv: { Values: (AnyArrayValue) }
      $fruits: { Values: (AssociativeArray UID: 2198) }
      $result: { Values: (unknown),(apple),(pear),(orange) }
      $value: { Values: (apple),(pear),(orange) }
      $isOrange: { Values: ('False' Type: Boolean),('True' Type:
Boolean) }

      ===GLOBAL CONTROLS===

      ===LOCAL CONTROLS===
      CTRL--$?: { Values: (UndefinedValue) }
      CTRL--$.staticVariables: { Values: (AssociativeArray UID: 133) }
      CTRL--$.analysisWarning: { Values: (UndefinedValue) }
      CTRL--$.analysisSecurityWarning: { Values: (UndefinedValue) }
      CTRL--$.catchBlocks: { Values:
(Weverca.Analysis.FlowResolver.TryBlockStack) }
      CTRL--$.constants: { Values: (AssociativeArray UID: 174) }
```

```
CTRL--$.staticVariableSink: { Values: (UndefinedValue) }
CTRL--$.evalDepth: { Values: ('0' Type: Int32) }


===ARRAYS===
$fruits[?]: { Values: (UndefinedValue) }
$fruits[0]: { Values: (apple) }
$fruits[1]: { Values: (pear) }
$fruits[2]: { Values: (orange) }

CTRL--$.staticVariables[?]: { Values: (UndefinedValue) }

CTRL--$.constants[?]: { Values: (UndefinedValue) }

TEMP--$24[?]: { Values: (UndefinedValue) }

TEMP--$31[?]: { Values: (UndefinedValue) }

TEMP--$42[?]: { Values: (UndefinedValue) }
TEMP--$42[0]: { Values: (apple) }
TEMP--$42[1]: { Values: (pear) }
TEMP--$42[2]: { Values: (orange) }

TEMP--$48[?]: { Values: (UndefinedValue) }
TEMP--$48[0]: { Values: (apple) }
TEMP--$48[1]: { Values: (pear) }
TEMP--$48[2]: { Values: (orange) }

TEMP--$54[?]: { Values: (UndefinedValue) }
TEMP--$54[0]: { Values: (apple) }
TEMP--$54[1]: { Values: (pear) }
TEMP--$54[2]: { Values: (orange) }



===FIELDS===

===ALIASES===
$_POST: {  MUST: $_POST | MAY }
$_GET: {  MUST: $_GET | MAY }
$_SERVER: {  MUST: $_SERVER | MAY }
$_COOKIE: {  MUST: $_COOKIE | MAY }
$_SESSION: {  MUST: $_SESSION | MAY }
$_FILES: {  MUST: $_FILES | MAY }
$_REQUEST: {  MUST: $_REQUEST | MAY }
```

```
      $GLOBALS: {  MUST: $GLOBALS | MAY }
      $_ENV: {  MUST: $_ENV | MAY }
      $php_errormsg: {  MUST: $php_errormsg | MAY }
      $argc: {  MUST: $argc | MAY }
      $argv: {  MUST: $argv | MAY }


Analysis completed in: 959ms


Analysis warnings:

No warnings

Security warnings:

No warnings
```

In example 4, we describe how `foreach` works. Compared to common `for` loop, `foreach` has one advantage: we know all values that are inside an array. We can initialize iteration variable so that it will contain all possible values from all indices of array. This simplification allows the analysis to go through `foreach` loop only once.

In the body of the first loop, we check whether the input from user is equal to an element of the array. Variable `$result` contains all values from the array and also previous value. It is correct, because condition with user input can succeed for every element or always fail and variable `$result` is not changed.

Inside the second loop, we test elements of the array for equality to a string literal. As we can see, the literal is stored in index 2 of the array. Nevertheless, the variable `$isOrange` contains both `TRUE` and `FALSE`. This is because the analysis compares the literal to a variable `$value` that has been initialized to multiple values. Therefore, this condition can be evaluated to any Boolean value. It is inaccuracy of the model caused by its simplification.

3.4 Weverca web

Weverca web can be found at http://perun.ms.mff.cuni.cz/weverca/. The user interface consists of a PHP code editor, checkboxes which tells the framework if we want to run verification and what metrics we want to run. After the analysis is finished, a page with results appears. There is again the analyzed code and several tabs with results. There are warnings in the first one, metrics results in the second one and all program points in the last one.

User can choose from several pre-entered PHP codes to analyze. User can also input custom code, or alter any of the pre-entered ones.

There are two memory models implemented in Weverca. Web interface lets user choose Memory Model for analysis. There is a timeout for the analyses to avoid server overload. It is currently set to 30 seconds.

4 References

[1] J. Clarke. SQL Injection Attacks and Defense, Second Edition. Syngress, July 2012.