BOA

Release 0.3

Cristian Garcia Romero

Jan 28, 2022

CONTENTS

BOA (Buffer Overflow Annihilator) is a vulnerability analyzer of general purpose. It is written in Python, and the main principle which it has coded has been to give the maximum flexibility to the user, and for that reason, modularity is a BOA's priority. Through dynamic module loading, it is possible to use the language parser which the user wants and use it to focus their own security needs.

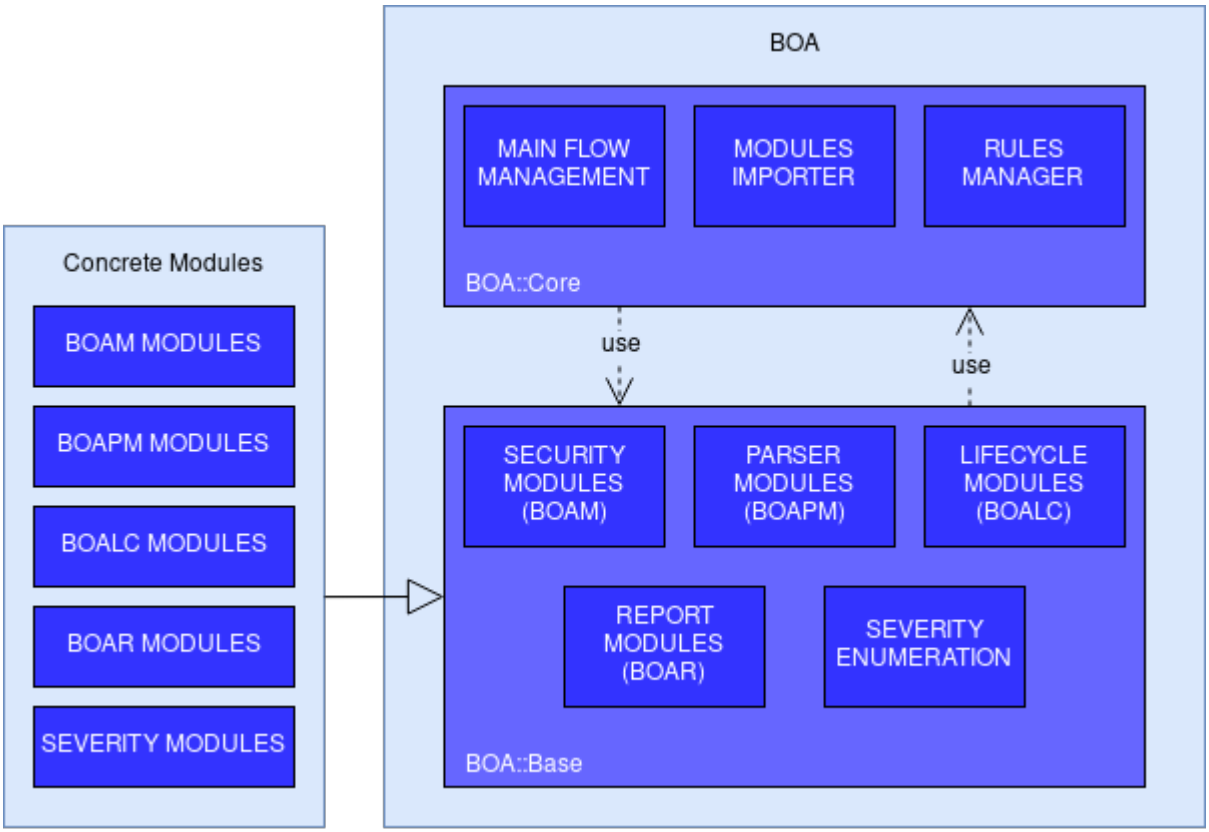

Fig. 1: BOA Architecture

CHAPTER

ONE

BOA MAIN FLOW

BOA main file.

This file handles the higher level interaction with BOA.

Main tasks:

- It handles args.
- It handles runners modules.
- It handles code modules (BOA's goal).
- It handles the general flow.

boa.**main**()
: It handles the main BOA's flow at a high level.

  Returns status code

  Return type int

boa.**manage_args**()
: It handles BOA's general args through ArgsManager class.

  Returns status code

  Return type int

CHAPTER

TWO

BOA INTERNALS

CHAPTER

THREE

BOA ARGUMENTS HANDLER

BOA arguments manager.

This file handles the arguments which are provided to BOA.

Concretely, the ArgsManager class loads, parses and checks the arguments. The arguments are defined in this file

class args_manager.**ArgsManager**

ArgsManager class.

It handles the arguments provided to BOA CLI.

__init__()

It creates the ArgParse parser object.

__weakref__

list of weak references to the object (if defined)

check()

It checks if the arguments are the expected type.

Concretely, in this method it is checked if the arguments are an instance of *argparse.Namespace*.

Returns status code

Return type int

load_args()

It creates the list of arguments.

parse()

It parses the arguments.

Returns status code

Return type int

CHAPTER

FOUR

CONSTANTS

File with constant values.

This file contains constant information about BOA.

It contains multiple classes:

1. Meta class: it contains information about BOA.
2. Error class: it contains status code with a descriptive name.
3. Regex class: it contains regex strings.
4. Other class: it contains all the constants whose goal does not match with the other classes.

class constants.**Error**

Error class.

It contains information about the different error status code we can find through BOA's code. The information that it is in this class are just numeric status error with a descriptive name to know exactly the cause of the error.

When BOA finishes the execution, it displays the status code. If the status code displayed matches with Meta.ok_code, it means that everything went fine. Otherwise, check the status code within this class.

class constants.**Meta**

Meta class.

It contains information about BOA like the version, the description, ...

class constants.**Other**

Other class.

This class contains all the other information that does not match with the goal of the other classes. Or does not have a concrete goal.

class constants.**Regex**

Regex class.

This class contains the regex which are used by other BOA modules.

CHAPTER

FIVE

RULES MANAGER

Rules Manager file.

This file contains the necessary methods in the RulesManager class to load, check and process the rules file.

The file rules should contain all the necessary information to execute a concrete analysis. However, multiple and independent analysis might be executed.

Each rules file should contain a complete analysis technique defined. If multiple modules are being used, this should be to reach the goal of execute a complex but concrete analysis.

class rules_manager.**RulesManager**(*rules_file*)

RulesManager class.

This class defines the necessary methods to load, check and process the rules from a file.

__init__(*rules_file*)

It initializes the necessary variables.

Parameters **rules_file** (*str*) – path to the rules file.

__weakref__

list of weak references to the object (if defined)

check_rules(*save_args*)

It checks if the rules file contains the mandatory rules. The checking is performed by name and elements quantity, so it has to match in both properties.

If *self.rules* is *None*, the checking fails.

Parameters **save_args** (*bool*) – it indicates that the arguments have to be saved while they are being checked.

Returns true if the rules are valid; false otherwise

Return type bool

check_rules_arg(*arg*, *father*, *grandpa*, *save_args*, *args_reference*, *sort_args*)

It checks if the <args> elements are correct recursevely.

This method works recursively making the following calls:

- check_rules_arg -> check_rules_arg_recursive
- check_rules_arg_recursive -> check_rules_arg

What this method makes is checking if the <args> elements which are inside are correct and, optionally, save them to being used by the target module which is specified in the rules file.

If the arguments want to be saved, the order may not be the expected. If the <args> elements are mixed, the predefined order will be:

1. Dictionaries
2. Lists
3. Elements

If you want to have the elements in the correct order which you wrote, you can avoid mixing the elements or use the *sort_args* argument to have a partial sorting (this partial sorting makes worse the performance while checking). We say "partial" because the result will be that if you are mixing elements, all them will be grouped and will appear in the order which the first type of element appeared.

Parameters

- **arg** – current arg which is being checked (processed recursively).
- **father** (*str*) – arg's father.
- **grandpa** (*str*) – arg's grandpa; father's father.
- **save_args** (*bool*) – it indicates if you want to save the args while they are being checked.
- **args_reference** – if *save_args* is *True*, this is the concrete arg reference which it changes while recursion is being processed. It changes the reference from call to call to save the concrete args here.
- **sort_args** (*bool*) – it indicates if you want a partial sorting when you mix different type of elements.

Example

<args>

<dict>

<list name="l1"></list>

<dict name="d"></dict>

<element name="e" value="v" />

<list name="l2"></list>

</dict>

</args>

Unsorted result: {"d": {}, "l1": [], "l2": [], "e": "v"}

Partial sorted result: {"l1": [], "l2": [], "d": {}, "e": "v"}

Returns true if the arguments are valid; false otherwise

Return type bool

check_rules_arg_high_level(*dict_tag*, *parent_tag_name*, *tag_prefix*, *save_args*)

This is the method that should be invoked when you want to check, and optionally parse, the arguments from the rules file.

Raises

- **BOARulesIncomplete** – when the number of expected mandatory rules does not match with the actual number of rules or when a concrete rule is not found.
- **BOARulesUnexpectedFormat** – when the format of a rule is not the expected.
- **BOARulesError** – when an non-specific error happens.

check_rules_arg_recursive(*arg*, *element*, *father*, *arg_reference*, *args_reference*, *save_args*, *sort_args*)
This method is used by *check_rules_arg* method.

This method wraps the common behaviour for saving the arguments references and makes the necessary recursive calls for each element. Moreover, it checks that the arguments, depending on the concrete element, contains the expected attributes (e.g. a dictionary's element contains a name attribute).

For details, check *check_rules_arg* documentation.

Parameters

- **arg** – current arg which is being checked (processed recursively).
- **element** (*str*) – the concrete type of element which is being given. It may be "dict", "list" or *None*. If *None*, it indicates that the element is a "element".
- **father** (*str*) – arg's father.
- **arg_reference** – the new reference which will be appended to *args_reference*.
- **args_reference** – if *save_args* is *True*, this is the concrete arg reference which it changes while recursion is being processed. It changes the reference from call to call to save the concrete args here.
- **save_args** (*bool*) – it indicates if you want to save the args while they are being checked.
- **sort_args** (*bool*) – it indicates if you want a partial sorting when you mix different type of elements.

Returns true if the arguments are valid; false otherwise

Return type bool

check_rules_dynamic_analysis_runner(*save_args*, *runner_module*)
It makes the checks relative to the runner modules which are used in the dynamic analysis.

Parameters

- **save_args** (*bool*) – it indicates that the arguments have to be saved while they are being checked.
- **runner_module** (*str*) – runner which is going to be processed in order to check the rules.

Raises

- **BOARulesIncomplete** – when the number of expected mandatory rules does not match with the actual number of rules or when a concrete rule is not found.
- **BOARulesError** – when a semantic rule is broken. You have to follow the rules documentation to avoid this exception.
- **KeyError** – if *runner_module* is not an expected runner.

check_rules_init()
It makes the initial checks.

Raises **BOARulesIncomplete** – when the number of expected mandatory rules does not match with the actual number of rules or when a concrete rule is not found.

check_rules_modules(*save_args*)
It makes the checks relative to the modules.

Parameters **save_args** (*bool*) – it indicates that the arguments have to be saved while they are being checked.

Raises

- **BOARulesIncomplete** – when the number of expected mandatory rules does not match with the actual number of rules or when a concrete rule is not found.
- **BOARulesUnexpectedFormat** – when the format of a rule is not the expected.
- **BOARulesError** – when an non-specific error happens.

check_rules_parser()
It makes the checks relative to the parser.

Raises **BOARulesIncomplete** – when the number of expected mandatory rules does not match with the actual number of rules or when a concrete rule is not found.

check_rules_report(*save_args*)
It makes the checks relative to the report.

Parameters **save_args** (*bool*) – it indicates that the arguments have to be saved while they are being checked.

Raises

- **BOARulesIncomplete** – when the number of expected mandatory rules does not match with the actual number of rules or when a concrete rule is not found.
- **BOARulesError** – when a semantic rule is broken. You have to follow the rules documentation to avoid this exception.

close()
It closes the rules file to release the resource.

Returns status code

Return type int

get_args(*module=None*)
It returns the args for a concrete module.

Parameters **module** (*str*) – module from which args are going to be returned. The expected format is (without quotes): "module_name.class_name". Check *utils.get_name_from_class_instance*. The default value is *None*.

Returns module args or all the modules args; *None* if a module were specified and could not find it

Return type dict

get_dependencies(*module=None*)
It returns the dependencies for a concrete module.

Parameters **module** (*str*) – module from which args are going to be returned. The expected format is (without quotes): "module_name.class_name". Check *utils.get_name_from_class_instance*. The default value is *None*.

Returns module dependencies or all the modules dependencies; *None* if a module were specified and could not find it, what means that the module does not have dependencies

Return type dict

get_report_args()
It returns the args for the Report instance.

Returns report args

Return type dict

get_rules(*path=None*, *list_type=False*)
It returns the rules.

The rules can be obtained with a concrete *path*, which means that you can obtain the rules you want directly, without go through them. The path is a string which will be splited with '.'.

Parameters

- **path** (*str*) – the returned rules will be from a starting point. If you to get all the modules rules, the path has to have the value "boa_rules.modules.module". The default value is *None*.
- **list_type** (*bool*) – the returned rules will be wrapped in a list. The default value is *False*.

Returns rules from the rules file. If *list_type* is *True*, the returning type will not be *dict* but *list*.

Return type dict

get_runner_args(*runner_module*)
It returns the args of a runner module.

Parameters **runner_module** (*str*) – key of *self.runner_args* which is where the parameters are stored.

Returns args if they exist, but empty dict and logging warning if does not

Return type dict

open()
It opens the rules file, checking if exists first.

Returns status code

Return type int

read()
It reads the rules file and saves the necessary information.

Returns status code

Return type int

set_args(*module*, *arg*)
It sets the arguments for a concrete module. This method should only be used for internal management.

Parameters

- **module** (*str*) – instance identification as string. The expected format is (without quotes): "module_name .class_name". Check *utils.get_name_from_class_instance*.
- **arg** (*dict*) – the new args for the module.

Returns *True* if the args could be set; *False* otherwise

Return type bool

set_dependencies(*module*, *dependencies*)
It adds dependencies for a concrete module. This method should only be used for internal management.

Parameters

- **module** (*str*) – instance identification as string. The expected format is (without quotes): "module_name .class_name". Check *utils.get_name_from_class_instance*.
- **dependencies** (*dict*) – the new dependencies for the module.

Returns *True* if the dependencies could be apppended; *False* otherwise

Return type bool

CHAPTER

SIX

MODULES IMPORTER

Module Imports.

This file contains the ModulesImporter class.

class modules_importer.**ModulesImporter**(*modules*, *filenames=None*)

ModulesImporter class.

This class has the goal of loading the modules which are specified in the given rules.

__init__(*modules*, *filenames=None*)

It initializes the class.

Parameters

- **modules** (*list*) – modules (str) which should be loaded after.
- **filenames** (*list*) – modules filenames (str). The default value is *None*, and if this value remains, *modules* will be used as filename.

__weakref__

list of weak references to the object (if defined)

get_instance(*module_name*, *class_name*)

It returns an instance of the class of a module.

Parameters

- **module_name** (*str*) – module name which should contains *class_name*.
- **class_name** (*str*) – class name which is attempted to return.

Returns Module instance if module is loaded; *None* otherwise

get_module(*module_name*)

It returns an already loaded module.

Parameters **module_name** (*str*) – module name which is attempted to return.

Raises

- **BOAModuleNotLoaded** – when it is attempted to get a module which is not loaded.
- **Exception** – when a module is detected as loaded but it is not loaded in *sys.modules*. It should not happen.

Returns Module if loaded; *None* otherwise

get_nloaded()

It returns the number of loaded modules at the moment of the calling.

Returns loaded modules

Return type int

`get_nmodules()`
It returns the number of modules which were suplied to the class to be loaded.

A different variable is being used instead of len() method because the variable which contains the methods to be loaded mutates through the execution of the class methods.

Returns initial modules to be loaded

Return type int

`get_not_loaded_modules()`
It returns the modules which have not been loaded.

Returns not loaded modules

Return type list

`is_module_loaded(module_name)`
It checks if a concrete module is already loaded.

Parameters **module_name** (*str*) – module to check if it is loaded.

Returns module_name is loaded

Return type bool

`load(module_subdir=None)`
It attempts to load all the modules which were specified.

This method iterates through self.modules to attempt to loading the modules. First, it checks if the module is already loaded. Then, it attempts to load the module and if it is not able to, it skips the current module to next.

The modules must be in *Other.modules_directory* directory.

Parameters **module_subdir** (*str*) – subdir of *Other.modules_directory* where the module will be looked for instead of directly look for in *Other.modules_directory*

`classmethod load_and_get_instance(module, absolute_file_path, class_name, verbose=True)`
Class method which attempts to load a module and return an instance of it.

If the module is already loaded, it will skip the loading part and it will continue to next phase: get the instance.

Parameters

- **module** (*str*) – module name to be loaded.
- **absolute_file_path** (*str*) – full path to the file which contains the module to be loaded.
- **class_name** (*str*) – class name inside the module which is going to be instantiated.
- **verbose** (*bool*) – if *True*, a message will be displayed if the loading success.

Returns an instance of "module.class" which has been specified or *None* if could not.

Return type instance

CHAPTER

SEVEN

EXCEPTIONS

CHAPTER

EIGHT

UTILITIES

CHAPTER

NINE

SEVERITY BASE (INTERFACE)

Severity levels.

In this file is defined the base enum for defining severity levels, whose can be overridden and new ones can be defined.

class enumerations.severity.severity_base.**SeverityBase**(*value*)

SeverityBase class (enum).

You can inherit from this class and implement your own severity levels. An approach could be to use the standard risk model (risk = likelyhood [0-N] * impact [0-N]). To allow the inheritance we have to let this enum empty.

The enum values mean the priority (higher means more critical). This values will be used for sorting the records.

CHAPTER

TEN

SEVERITY SYSLOG

Severity level based on syslog (RFC 5424).

This severity level defines 8 levels of severity.

class enumerations.severity.severity_syslog.**SeveritySyslog**(*value*)

SeveritySyslog class (enum).

There are multiple ways of defining severity levels. For the severity base we are using an approach based on Syslog Message Severities (RFC 5424).

CHAPTER

ELEVEN

SEVERITY FOR MODULE BOAMODULEFUNCTIONMATCH

Severity level for the BOAModuleFunctionMatch.

This severity level defines 3 levels of severity.

class enumerations.severity.severity_function_match.**SeverityFunctionMatch**(*value*)
SeverityFunctionMatch enum.

CHAPTER

TWELVE

SEVERITY ENUMS

CHAPTER

THIRTEEN

OTHER

- *Severity for Module BOAModuleFunctionMatch*

CHAPTER

FOURTEEN

MODULES

CHAPTER

FIFTEEN

OTHER

- *Changelog*
- *TODO List*

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SEVENTEEN

AUXILIARY MODULE - PYCPARSER AST PREORDER VISITOR

CHAPTER

EIGHTEEN

AUXILIARY MODULE - PYCPARSER CFG

CHAPTER

NINETEEN

AUXILIARY MODULE - PYCPARSER UTIL

CHAPTER

TWENTY

AUXILIARY MODULES

CHAPTER

TWENTYONE

MODULES

CHAPTER

TWENTYTWO

OTHER

- *Changelog*
- *TODO List*

CHAPTER

TWENTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

TWENTYFOUR

MAIN MODULES

The modules which BOA uses for making the core works.

24.1 Modules

- *BOA main flow*
  - BOA main flow. It is the entry point.
- *BOA internals*
  - It has methods which BOA uses in the main flow.
- *BOA arguments handler*
  - It is an utility which works with ArgParse and helps us to manage the BOA args.
- *Constants*
  - Constant values definitions.
- *Rules Manager*
  - It handles the rules files checking they are well formatted and it gives us the information.
- *Modules Importer*
  - It handles the modules importing, focusing the security modules. It has utilities for importing other modules.
- *Severity Enums*
  - It contains enumerations which defines different levels of severity.
- *Exceptions*
  - It defines own exceptions.
- *Utilities*
  - General utilities.
- *Auxiliary Modules*
  - It contains auxiliary modules which may be used by any other module.

CHAPTER

TWENTYFIVE

MODULES

CHAPTER

TWENTYSIX

OTHER

- *Changelog*
- *TODO List*

CHAPTER

TWENTYSEVEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

TWENTYEIGHT

BOAMODULEABSTRACT

CHAPTER

TWENTYNINE

BOAM - FUNCTION MATCH

CHAPTER

THIRTY

BOAM - CFG

CHAPTER

THIRTYONE

BOAM - CFG

CHAPTER

THIRTYTWO

BOAM - TEST

CHAPTER

THIRTYTHREE

SECURITY MODULES

Security modules are the main way a user can define its own modules in order to look for a concrete threat.

These modules can be found in the main directory of BOA, concretely in the directory "modules". The files you will find there will have a name like "boam_whatever.py", but is not necessary to follow the nomenclature. You can name your modules as you like. If you want to write your own, you will have to store your module in the expected directory (i.e. /path/to/BOA/modules).

All your security modules will need to inherit from *BOAModuleAbstract* in order to work as a security module.

33.1 BOA internals

- *BOAModuleAbstract*

33.2 Modules

- *BOAM - Function Match*
- *BOAM - CFG*
- *BOAM - Test*
- *BOAM - CFG*

CHAPTER

THIRTYFOUR

MODULES

CHAPTER

THIRTYFIVE

OTHER

- *Changelog*
- *TODO List*

CHAPTER

THIRTYSIX

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THIRTYSEVEN

BOALIFECYCLE MANAGER

Lifecycle manager.

This file contains the class BOALifeCycleManager, which handles the loop which is executed to analyze the language file.

class lifecycles.boalc_manager.**BOALifeCycleManager**(*instances*, *reports*, *lifecycle_args*, *lifecycle_instances*, *analysis*)

BOALifeCycleManager class.

This class handles the modules intances. Concretely, it initializes the instances, iterates the processing throught them, and save the report records.

The steps which are followed depends on the lifecycle being used by a concrete module.

__init__(*instances*, *reports*, *lifecycle_args*, *lifecycle_instances*, *analysis*)

It initializes all the variables which will be used by the other methods.

Parameters

- **instances** (*list*) – module instances that are going to be saved.
- **reports** (*list*) – list of Report instances.
- **lifecycle_args** (*dict*) – args to be used by the lifecycles.
- **lifecycle_instances** (*list*) – instances of lifecycles to be used by the *instances*.
- **analysis** (*str*) – information about which analysis we are running.

__weakref__

list of weak references to the object (if defined)

execute_instance_method(*instance*, *method_name*, *args*, *force_invocation*)

It attempts to execute a method of a concrete instance.

Parameters

- **instance** – initialized instance which a method is going to be invoked if possible.
- **method_name** (*str*) – method which is going to be invoked.
- **args** – args to be given to the invoked method.
- **force_invocation** (*bool*) – force a method invocation despite something failed in the past (if *False*, when a failure happens, a method will not be invoked).

Returns it will return *False* if: the instance is not in *self.instances*, the property *stop* is *True*, ... It will return *True* only if the execution of the given method could be executed without any exception.

Return type bool

`get_final_report()`
It returns the final report.

Returns the final report

Return type Report

`handle_lifecycle()`
This method is the one which should be invoked to handle the lifecycle.

The way the phases are invoked depends on the lifecycle.

The method that will be invoked is *execute_lifecycle*, which is defined in *BOALifeCycleAbstract* class. In that method should be defined the phases that are going to be called.

Returns self.rtn_code

Return type int

`make_final_report()`
It makes a report which contains all the threat records contained in all the other reports.

Returns the final report or *None*

Return type Report

CHAPTER

THIRTYEIGHT

BOALIFECYCLE ABSTRACT

This file contains the class from which all lifecycles will have to inherit in order to be executed as a lifecycle. If a lifecycle does not inherit from the implemented class in this file, an error will be raised.

class lifecycles.boalc_abstract.**BOALifeCycleAbstract**(*instance*, *report*, *lifecycle_args*, *execute_method_callback*, *analysis*)

BOALifeCycleAbstract class.

This class implements the necessary methods which will be invoked after by *BOALifeCycleManager*. Moreover, it defines variables with important information (e.g. "args" variable which contains the given arguments throught the rules file).

__init__(*instance*, *report*, *lifecycle_args*, *execute_method_callback*, *analysis*)

It initializes the class.

Parameters

- **instance** – initialized instance to be invoked.
- **report** (BOAReportAbstract) – report where add found threats.
- **lifecycle_args** (*dict*) – args to be used by the lifecycle.
- **execute_method_callback** (*func*) – function which will be executed in order to execute *instance*.
- **analysis** (*str*) – information about which analysis we are running.

__weakref__

list of weak references to the object (if defined)

abstract execute_lifecycle()

Method which defines the concrete lifecycle to be executed. This method will have to be implemented by those lifecycles which want to define a new lifecycle.

get_name()

Method which returns the name of the concrete instance.

This method is defined because a security module can only access method by name (it invokes the methods by a method which is given by a callback) and no directly access to the variables.

This method could be invoked by a security modules in order to, for example, give a concrete error message.

Returns self.who_i_am (i.e. '"module_name"."class_name"')

Return type str

abstract raise_exception_if_non_valid_analysis()

Method which will be executed when a lifecycle has been initialized and should raise an exception if the analysis is not valid for the lifecycle.

Raises `BOALCAnalysisException` – when the selected analysis is not compatible with the life-cycle.

CHAPTER

THIRTYNINE

BOALC - BASIC

CHAPTER

FORTY

BOALC - PYCPARSER AST

CHAPTER

FORTYONE

LIFECYCLES

These modules are the ones which defines the way the execution is driven. When you want to perform a concrete execution of your security modules, a lifecycle might do what you want do. In the lifecycles are defined the methods and the order in which they will be invoked, and you can take the decision of what information will need your security modules, use callbacks as arguments to make a call back to a method of your lifecycle and take that feedback to your lifecycle for taking decisions, etz.

These modules can be found in the main directory of BOA, concretely in the directory "lifecycles". The files you will find there will have a name like "boalc_whatever.py", but is not necessary to follow the nomenclature. You can name your modules as you like. If you want to write your own, you will have to store your module in the expected directory (i.e. /path/to/BOA/lifecycles).

All your lifecycles will need to inherit from *BOALifeCycleAbstract* in order to work as a lifecycle.

41.1 BOA internals

- *BOALifeCycle Manager*
- *BOALifeCycle Abstract*

41.2 Modules

- *BOALC - Basic*
- *BOALC - Pycparser AST*

CHAPTER

FORTYTWO

MODULES

CHAPTER

FORTYTHREE

OTHER

- *Changelog*
- *TODO List*

CHAPTER

FORTYFOUR

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FORTYFIVE

BOAPARSERMODULEABSTRACT

CHAPTER

FORTYSIX

BOAPM - PYCPARSER

CHAPTER

FORTYSEVEN

PARSER MODULES

The parser modules are the one which works directly with a parser in order to process a code file and get processed data structures. With these datastructures you will be able to work in your security modules.

You should have a parser module for each programming language you want to work with, but can have different parser for the same programming language if you like.

These modules can be found in the main directory of BOA, concretely in the directory "parser_modules". The files you will find there will have a name like "boapm_whatever.py", but is not necessary to follow the nomenclature. You can name your modules as you like. If you want to write your own, you will have to store your module in the expected directory (i.e. /path/to/BOA/parser_modules).

All your parser modules will need to inherit from *BOAParserModuleAbstract* in order to work as a parser module.

47.1 BOA internals

- *BOAParserModuleAbstract*

47.2 Modules

- *BOAPM - Pycparser*

CHAPTER

FORTYEIGHT

MODULES

CHAPTER

FORTYNINE

OTHER

- *Changelog*
- *TODO List*

CHAPTER

FIFTY

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FIFTYONE

BOAREPORTABSTRACT

This file contains the class which is the base for the Report, which is the class that displays the information about all the found threats.

class reports.boar_abstract.**BOAReportAbstract**(*severity_enum*, *args*)

BOAReportAbstract class.

It implements the necessary methods to initialize, fill and display the threats report after the analysis.

If you want to define your own Report class you will have to define a new class which inherits from this one.

Raises **BOAReportException** – this exception could be raised anywhere in the class.

__init__(*severity_enum*, *args*)

It initializes the class with the necessary variables.

Parameters **severity_enum** (*type*) – enumeration which will be used for the threats severity. It has to inherit from *SeverityBase* but not be *SeverityBase*.

Raises

- **BOAReportEnumTypeNotExpected** – when *severity_enum* is not a type of *SeverityBase* or is *SeverityBase*.
- **TypeError** – when *severity_enum* is not a type or at least not an expected instance.

__weakref__

list of weak references to the object (if defined)

add(*who*, *description*, *severity*, *advice=None*, *row=None*, *col=None*, *sort_by_severity=True*, *severity_enum=None*)

It adds a new record to the main report.

Parameters

- **who** (*str*) – "module_name.class_name" (without quotes) format to identify who raised the threat.
- **description** (*str*) – description about the found threat.
- **severity** (SeverityBase) – threat severity.
- **advice** (*str*) – advice to solve the threat. It is optional.
- **row** (*int*) – threat row. It is optional.
- **col** (*int*) – threat col. It is optional.
- **sort_by_severity** (*bool*) – if *True*, the threats will be added sorting by severity (higher values will be added first). The default value is *True*.

- **severity_enum** (*type*) – enumeration which will be used for the threats severity. This arg is intended to be able to join different Report instances. Default is *None* which means to use *self.severity_enum*.

Returns status code

Return type int

append(*report_instance*, *sort_by_severity=True*, *stop_if_fails=False*, *who=None*)
It appends other threats report records to this.

The goal of this method is to be able to append multiple reports which will be created for each module and end up with only a report to show to the user.

Parameters

- **report_instance** (*Report*) – the report to be appended to this.
- **sort_by_severity** (*bool*) – if *True*, the threats will be added sorting by severity (higher values will be added first). The default value is *True*.
- **stop_if_fails** (*bool*) – if *True* and any threat record cannot be appended, the execution will stop. The default value if *False*.
- **who** (*str*) – the module name which is going to be used to set the relation between the module and the report instance.

Returns status code

Return type int

abstract display(*who*, *display=True*)
It displays all the threats from a concrete module.

This method is intended to be invoked by *display_all*.

Parameters

- **who** (*str*) – the module which found the threat.
- **display** (*bool*) – if *True*, it displays the threat.

Raises **BOAReportWhoNotFound** – if the given module is not found.

Returns text to be displayed

Return type str

abstract display_all(*print_summary=True*, *display=True*)
It displays all the threats from all the modules. Moreover, it prints a summary at the end optionally.

This method should invoke *display* which should invoke *pretty_print_tuple*. You can avoid this overriding the methods using "pass", but if you do this, this method will have to make all the work.

Parameters

- **print_summary** (*bool*) – if *True*, it prints a summary with statistics about all the found threats.
- **display** (*bool*) – if *True*, it displays the threat.

Returns text to be displayed

Return type str

get_severity_enum_instance()
It returns the severity enumeration instance which is being used.

Returns severity enumeration being used

Return type *SeverityBase*

Note: This is the **GENERAL** severity enum reference, which may not be what you are looking for. If you want the severity enum instance of a concrete module, use *get_severity_enum_instance_by_who()* instead.

`get_severity_enum_instance_by_who`(*who*)

It returns the severity enum instance of a concrete module.

Parameters **who** (*str*) – module name in format "module_name.class_name".

Returns the severity enum instance which is used for the given module. *None* if *who* is not found

Return type *SeverityBase*

`get_summary`()

It returns a summary of all the threat records.

Returns summary of threat records. Its key format is (without quotes) "module_name.class_name" and the value is a list of tuples

Return type dict

`get_who`()

It returns the modules which are in the current report.

Returns list containing the modules which are in the current report

Return type list (str)

`abstract pretty_print_tuple`(*t*, *first_time=False*, *reported_by=False*, *display=True*)

It prints a pretty line about a found threat record.

This method is intended to be invoked by *display*.

The expected format for the tuple is next:

1. str: module who raised the threat.
2. str: threat description.
3. SeverityBase: threat severity.
4. str (optional): advice for solving the threat. If it is not provided, the string "not specified" will be displayed.
5. int (optional): threat row. If it is not provided, the value -1 will be displayed.
6. int (optional): threat col. If it is not provided, the value -1 will be displayed.
7. **type: SeverityBase type which will be used to display** the severity. This value is intented to be able to join different Report instances.

Parameters

- **t** (*tuple*) – threat record.
- **first_time** (*bool*) – if you want to display a pretty box around the module name who raised the threat, this value must be *True*. The default value is *False*.
- **reported_by** (*bool*) – if you want to display the module who raised the threat, this value must be *True*. This arg should be used when you want to avoid the arg *first_time*. The default value is *False*.

- **display** (*bool*) – if *True*, it displays the threat.

Returns text to be displayed

Return type str

Note: If you want to show orderly the threats, you should use *first_time=True* for the first record and *first_time=False* for the rest. If you do not want to show it orderly, you should use *reported_by=True*.

---

set_severity_enum_mapping(*who*, *severity_enum_instance*)
It sets the relation between a module and a severity enum.

Parameters

- **who** (*str*) – the module name in format "module_name.class_name".
- **severity_enum_instance** (SeverityBase) – severity enum instance.

Returns it returns *True* if the relation was set. *False* otherwise

Return type bool

CHAPTER

FIFTYTWO

BOAR - STDOUT

This file contains the BOARStdout class, which inherits from the abstract Report class. This base has the goal of report the found threats using the standard output. It is a basic way of report the threats.

class reports.boar_stdout.**BOARStdout**(*severity_enum*, *args*)

BOARStdout class.

It implements the necessary methods to initialize, fill and display the threats report after the analysis.

display(*who*, *display=True*)

It displays all the threats from a concrete module.

Parameters

- **who** (*str*) – the module which found the threat.
- **display** (*bool*) – if *True*, it displays the threat.

Raises **BOAReportWhoNotFound** – if the given module is not found.

Returns text to be displayed

Return type str

display_all(*print_summary=True*, *display=True*)

It displays all the threats from all the modules. Moreover, it prints a summary at the end optionally.

Parameters

- **print_summary** (*bool*) – if *True*, it prints a summary with statistics about all the found threats.
- **display** (*bool*) – if *True*, it displays the threat.

Returns text to be displayed

Return type str

pretty_print_tuple(*t*, *first_time=False*, *reported_by=False*, *display=True*)

It prints a pretty line about a found threat record.

The expected format for the tuple is next:

1. str: module who raised the threat.
2. str: threat description.
3. SeverityBase: threat severity.
4. str (optional): advice for solving the threat. If it is not provided, the string "not specified" will be displayed.
5. int (optional): threat row. If it is not provided, the value -1 will be displayed.

6. int (optional): threat col. If it is not provided, the value -1 will be displayed.
7. **type: SeverityBase type which will be used to display** the severity. This value is intented to be able to join different Report instances.

Parameters

- **t** (*tuple*) – threat record.
- **first_time** (*bool*) – if you want to display a pretty box around the module name who raised the threat, this value must be *True*. The default value is *False*.
- **reported_by** (*bool*) – if you want to display the module who raised the threat, this value must be *True*. This arg should be used when you want to avoid the arg *first_time*. The default value is *False*.
- **display** (*bool*) – if *True*, it displays the threat.

Returns text to be displayed

Return type str

Note: If you want to show orderly the threats, you should use *first_time=True* for the first record and *first_time=False* for the rest. If you do not want to show it orderly, you should use *reported_by=True*.

CHAPTER

FIFTYTHREE

BOAR - BASIC HTML

This file contains the implementation of the necessary methods of the report abstract class. This report class uses HTML files to report about the found threats.

class reports.boar_basic_html.**BOARBasicHTML**(*severity_enum*, *args*)

BOARBasicHTML class.

It implements the necessary methods to alert about the found threats and interact with the HTML file.

display(*who*, *display=True*)

It creates the HTML table for a concrete module.

Parameters

- **who** (*str*) – the module which found the threat.
- **display** (*bool*) – if *True*, it displays the threat.

Raises **BOAReportWhoNotFound** – if the given module is not found.

Returns text to be displayed in HTML format

Return type str

display_all(*print_summary=True*, *display=True*)

It displays all the threats from all the modules. Moreover, it prints a summary at the end optionally. All in HTML format.

Parameters

- **print_summary** (*bool*) – if *True*, it prints a summary with statistics about all the found threats.
- **display** (*bool*) – if *True*, it displays the threat.

Returns text to be displayed in HTML format.

Return type str

pretty_print_tuple(*t*, *first_time=False*, *reported_by=False*, *display=True*)

It prints a pretty line about a found threat record.

The expected format for the tuple is next:

1. str: module who raised the threat.
2. str: threat description.
3. SeverityBase: threat severity.
4. str (optional): advice for solving the threat. If it is not provided, the string "not specified" will be displayed.

5. int (optional): threat row. If it is not provided, the value -1 will be displayed.
6. int (optional): threat col. If it is not provided, the value -1 will be displayed.
7. **type: SeverityBase type which will be used to display** the severity. This value is intented to be able to join different Report instances.

Parameters

- **t** (*tuple*) – threat record.
- **first_time** (*bool*) – if you want to display a pretty box around the module name who raised the threat, this value must be *True*. The default value is *False*.
- **reported_by** (*bool*) – if you want to display the module who raised the threat, this value must be *True*. This arg should be used when you want to avoid the arg *first_time*. The default value is *False*.
- **display** (*bool*) – if *True*, it displays the threat.

Returns text to be displayed in HTML format

Return type str

save_html(*inner_html*)

It saves the HTML content in the expectified file.

Args "absolute_path" and "filename" has to be defined in the rules file.

Parameters **inner_html** (*str*) – HTML content.

CHAPTER

FIFTYFOUR

REPORTS

The reports are the last phase of BOA and is where all the found threats are displayed. The way the reports can be displayed is customizable. There are basic Report implementation like *BOARStdout* or *BOARBasicHTML*, but you can define your own.

These modules can be found in the main directory of BOA, concretely in the directory "reports". The files you will find there will have a name like "boar_whatever.py", but is not necessary to follow the nomenclature. You can name your modules as you like. If you want to write your own, you will have to store your module in the expected directory (i.e. /path/to/BOA/reports).

All your report modules will need to inherit from *BOAReportAbstract* in order to work as a report module.

54.1 BOA internals

- *BOAReportAbstract*

54.2 Modules

- *BOAR - Stdout*
- *BOAR - Basic HTML*

CHAPTER

FIFTYFIVE

MODULES

CHAPTER

FIFTYSIX

OTHER

- *Changelog*
- *TODO List*

CHAPTER

FIFTYSEVEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

FIFTYEIGHT

CHANGELOG

You will find here the main changes from one version to others.

58.1 Version 0.4

Minnor changes.

Changes:

- Some error codes has been updated.
- When input arguments are not correctly inserted, now not only argparse displays error message, but also BOA.

Fixed errors:

- When no environment variables was being used for a parser module in the rules file, the running analysis crashed.

58.2 Version 0.3

Dependencies among modules are possible.

Changes:

- The results of a module can be a dependency for others.

Fixed errors:

- Main argument "-\-no-fail" was not working as expected.
- When a module loading failed and the execution continued, was not being correctly removed.
- Minnor fixes.

58.3 Version 0.2

This version has made other elements to be customizable.

Changes:

- **Support for other programming lenguages.**
  - Customizable parser modules.
- Customizable lifecycles.

- Customizable reports.

Fixed errors:

- When a module could not load properly its arguments, was smashing all the arguments of the other modules.
- Some checks were not being done to avoid that customizable elements did not inherit from the defined abstract class.

58.4 Version 0.1

This version has finished BOA core implementation.

Changes:

- **Support for C programming language (with pycparser).**
  - Support only for AST.
- **Rules files parsing.**
  - Very flexible with arguments for modules.
- Unique lifecycle.
- Multiple modules execution.
- Modules customizable.
- **Threats report.**
  - Severity customizable.

CHAPTER

FIFTYNINE

MODULES

CHAPTER

SIXTY

OTHER

- *Changelog*
- *TODO List*

CHAPTER

SIXTYONE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SIXTYTWO

TODO LIST

Here you will find a list of ideas to be implemented in BOA. It is not sure that these ideas will be implemented.

- Custom reports (implemented in version 0.2).
- Results from a module as args for other (implemented in version 0.3).
- Defer a module execution.
- Give the user the possibility of disable a false-positive detection.

CHAPTER

SIXTYTHREE

MODULES

CHAPTER

SIXTYFOUR

OTHER

- *Changelog*
- *TODO List*

CHAPTER

SIXTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SIXTYSIX

MODULES

CHAPTER

SIXTYSEVEN

OTHER

- *Changelog*
- *TODO List*

CHAPTER

SIXTYEIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

P

R

S