DESIGN OF ALGORITHMS USING PYTHON 2021/2022

Discovering security vulnerabilities in Python programs

Checkpoint 3 on pages 14-20

1 Introduction

Software security vulnerabilities are weaknesses that are encoded in the software and which can be exploited by an attacker to achieve illegal access to resources. They are a widespread and growing global problem. In 2021 alone 22000 software vulnerabilities were detected.

A large class of vulnerabilities in applications, generically known as *code injections*, originates in programs that enable user input information to affect the values of certain parameters of security sensitive functions during execution. The problem is that if the user is an attacker, he can contrive special inputs that cause these functions to behave in unexpected ways - like for example enabling the attacker to gain control over the computer system!

These programs can be seen to encode a potentially dangerous information flow, in the sense that untrusted information inserted by an attacker via input functions (known as *sources*) may reach the arguments of sensitive functions or variables (so called *sinks*) and induce the program to perform security violations. For this reason, such flows can be deemed *illegal* for their potential to encode vulnerabilities.

It is possible to process program's input in order to eliminate any dangerous content it might have. This is performed by special functions (known as *sanitizers*), which can in practice neutralize the potential vulnerability, or validate that there is none. However, such techniques are often forgotten by programmers, who unwillingly leave vulnerabilities in the program that end up in the final software product. Furthermore, malicious programs can be written that encode vulnerabilities in an obscure way that is hard to detect.

The aim of this project is to develop a tool for detecting vulnerabilities in Python programs by tracing illegal flows from sources to sinks which are not cleaned by sanitizers. We will focus on a small subset of the language, but the same principles can be (and are) applied to the full language just as well.

There will be two intermediate checkpoints that will guide the construction of the program by breaking it into simpler tasks, and by defining useful abstract data-types that enable to reason more easily about the more advanced components of the program at a later stage.

The project is an opportunity for you to practice programming in Python and to apply your knowledge on some data structures and algorithms learned in the course. As a side benefit you can learn about the inner-workings of the Python language, and about software security principles that programmers should be aware of.

1.1 Examples

To see how code injections works in practice, consider this simple Python example code that implements a basic calculator. The calculator computes the result of the input provided by the user, which expects expressions written in Python.

```
comp = input('\nPlease insert an expression: ')
print ("Result =", eval(comp))
```

An example of a possible interaction could be:

```
Please insert an expression: 30 * 12 + 5
Result = 365
```

However, instead of writing an arithmetic expression, a user could insert the following valid Python expression

```
__import__('os').system('rm -rf /')
```

The `__import__` function dynamically imports the module named by the string provided, so this invokes the standard `os.system` function that invokes a shell to execute the given command (`rm -rf /`) which removes the file system root if the process has sufficient privileges. However, the user input could be validated (sanitized) before being passed to `eval`:

```
comp = input('\nPlease insert an expression: ')
if validate(comp):
   print ("Result =", eval(comp))
else:
   print ("Error")
```

A real world example of a vulnerability called an SQL Injection, that is prevented by the sanitizer `escape` is captured by the following lines:

```
f_tmstmp = request.args.get('f_timestmp', None)
new_q_str = 'SELECT * FROM insphre AND a_tmstmp >= ' + f_tmstmp
query_string = flask.escape(new_q_str)
it = engine.execute(query_string)
```

Notice that in both examples, the underlying problem is that the user input, received respectively by functions `input` and `request.args.get` (sources) reaches the arguments of sensitive functions `eval` and `engine.execute` (sinks). Functions `validate` and `escape` (sanitizers) are able to eliminate the vulnerability.

2 Overall functionality of the tool

The tool to be developed should be able to provide information about vulnerabilities that are encoded in a Python program. It will therefore receive as input a Python program, and information about vulnerability patterns that should be detected.

To make the problem more tractable, instead of dealing with complete Python programs, which can consist of many modules, classes and functions, we will assume that the code to be analyzed has undergone a pre-processing stage that has extracted a selection of Python instructions that capture any relevant information flows, and which we call code *slices*.

The vulnerability patterns specify for a given type of vulnerability its possible sources, sanitizers and sinks:

- name of vulnerability (e.g., `SQL injection`)
- a set of sources, the functions that serve as entry points (e.g., `get`),
- a set of sanitizers, the functions that can neutralize this vulnerability (e.g., `escape`),
- a set of sinks, the security sensitive functions that need to be protected (e.g., `execute`)

For each given pattern, the tool should warn about all data flows encoded in the program that originate in a source, reaches a sink, and that has not passed through any sanitizer. If it identifies a possible data flow from a source entry point to a sensitive sink (according to the inputted patterns), it should report a potential vulnerability; if the data flow passes through a corresponding sanitization function, it should not report it.

2.1 JSON format

Section 3 specifies the behaviour of the tool in more detail, including the format of the input and output that the tool should produce. Some of the data will be treated in the JSON format. This roughly means that it is represented in plain text, structured into *lists* '`[...]`', *objects* '`{...}`' containing pairs '*string*`:`*value*', and values.

Data in JSON is easily converted into corresponding structures in Python as expected (dictionaries, lists, etc). It can be done by calling `json.loads(json_data)` from the `json` module, and , which returns the representation of the data using Python structures.

Vulnerability patterns. The following example illustrates the representation in JSON of a list of vulnerability patterns that contain three patterns as JSON objects:

```
[
  {"vulnerability": "SQL injection A",
  "sources": ["get", "get_object_or_404", "QueryDict", "ContactMailForm"],
  "sanitizers": ["mogrify", "escape_string"],
  "sinks": ["execute"]},

  {"vulnerability":"SQL injection B",
```

```
  "sources":["QueryDict", "ContactMailForm", "ChatMessageForm", "copy"],
  "sanitizers":["mogrify", "escape_string"],
  "sinks":["raw", "RawSQL"]},

  {"vulnerability":"XSS",
  "sources":["get", "get_object_or_404", "QueryDict", "ContactMailForm"],
  "sanitizers":["clean","escape","flatatt","render_template","render"],
  "sinks":["send_mail_jinja","mark_safe","Response","Markup"]}
]
```

Program slices and ASTs. Even though the program slice to be analysed is received as Python code in text format, the very first step your tool should perform is to convert it to an Abstract Syntax Tree represented in JSON. To do so, you can use the tool `astexport`[1].

Until the third stage of the project, you will not yet need to analyse the ASTs. We have seen in class what they are, and we will revise ASTs in more detail when we study the Tree data type. Meanwhile, you can make yourself familiar with its structure. Have in mind that only a small selection of Python's program constructs will be considered (function calls, assignments, conditions and while loops), and that not all of the information that is available in the AST is necessarily relevant for this project.

The AST has the same structure as in Python's AST module[2]. To explore the structure of the AST that corresponds to a given program you can use this Python AST explorer[3]. This tutorial[4] is a helpful resource. If you would like to inspect data represented in JSON, you can visualize it using this online tool[5]. Note that the structure of Python's ASTs varies slightly with different Python versions. The examples given use Python 3.9 ASTs – as in the labs, similar to 3.8 and 3.10.)

Please note that while Python's `ast` module is dedicated to the analysis of ASTs, it is *not* to be used in this project.

3 Specification of the tool – Checkpoint 1

The aim of this first stage is to set up the basic abstract data types that your program will operate with, and to build the basic interaction cycle for providing inputs, and executing commands.

3.1 Commands

The tool should be callable in the command line, with no arguments:

```
$ python ./mytool.py
```

[1] You can install it from `https://pypi.org/project/astexport/`
[2] `https://docs.python.org/3.10/library/ast.html`
[3] `https://python-ast-explorer.com/`
[4] `https://greentreesnakes.readthedocs.io/en/latest/`
[5] `http://jsonviewer.stack.hu/`

It should then enter a cycle where it waits for a new command and executes it, until a command for exiting the program is given. The possible commands are the following: These commands

`p file_name`	read a new Python program slice to analyse from file `file_name`
`b file_name`	read new base vulnerability patterns from file `file_name`
`e json_pattern`	extend base vulnerabilities with `json_pattern`
`d vuln_name`	delete vulnerability pattern `vuln_name` from base
`c`	show current program slice and vulnerability patterns
`x`	exit the program

Table 1: Commands for running the tool.

should be read from the standard input, and the result printed to the standard output, according to the following behavior specification.

Command `p file_name`**.** The file `file_name` should be a Python program. The tool should convert it to an *Abstract Syntax Tree* (AST) represented in JSON, and then import it into Python. The imported AST should be stored, and replace any previously imported ASTs. At this stage of the project no further processing of the program slice is performed.

Command `b file_name`**.** The file `file_name` should contain a list of vulnerability patterns represented in JSON. The tool should import it into Python, and store it as a new base of vulnerability patterns, replacing any previously imported patterns. It should ensure that all the vulnerability patterns that are stored in the base have unique names.

Command `e json_pattern`**.** The argument `json_pattern` should be a JSON object representing a vulnerability pattern. The tool should import it into Python, and add it to the current base of vulnerability patterns. It should ensure that all the vulnerability patterns that are stored in the base have unique names.

Command `d vuln_name`**.** The argument `d vuln_name` should be a vulnerability name. The tool should remove from the base of vulnerability patterns a vulnerability with that name.

Command `c`**.** The tool should print the currently stored program slice, as well as the vulnerability pattern base, by alphabetical order of vulnerability names.

If there is an error in the process, the program should raise an exception with an appropriate error message. You can however assume that the types of the arguments are correctly introduced by the user (so you do not need to validate the types of the arguments).

3.2 Abstract Data Types

Your tool should include and use an implementation of the abstract data types Pattern, Flow, Policy and Label with the interface defined below.

1. To make sure that your program respects the abstraction barriers of the ADTs, **all field attributes of these classes should be hidden**, i.e., should be named using the double underscore: `__field_name`.

2. Document all of the classes and methods in order to make explicit: the **internal representation** of the objects of each class, as well as the **parameter and return types** of each method.

Pattern. The Pattern ADT represents a vulnerability pattern, including all its components, and is used to define Policies. It includes the following methods:

- `__init__ : Pattern × dict →`

  Receives a new Pattern object `self` and a dictionary representation of a pattern imported from JSON, and initializes it (`self` is automatically returned at instantiation time).

- `get_vulnerability : Pattern → str`

  Receives the Pattern object `self`, and returns its vulnerability name as a string.

- `get_sanitizers : Pattern → list(str)`

  Receives the Pattern object `self`, and returns its list of sanitizers as strings.

- `is_source : Pattern × str → bool`

  Receives the Pattern object `self` and a name as a string, and returns the value `True` or `False` indicating whether the name is a source of `self`.

- `is_sanitizer : Pattern × str → bool`

  Receives the Pattern object `self` and a name as a string, and returns the value `True` or `False` indicating whether the name is a sanitizer of `self`.

- `is_sink : Pattern × str → bool`

  Receives the Pattern object `self` and a name as a string, and returns the value `True` or `False` indicating whether the name is a sink of `self`.

- At least the standard Python method `__str__`.

Flow. The Flow ADT is used to represent an information flow from a source. It is used to build Labels that represent which flows might have influenced a certain piece of data. The Flow ADT includes the following methods:

- `__init__ : Flow × str →`

  Receives a new Flow object `self` and a source name as a string, and initializes it (it is automatically returned at instantiation time).

- `get_source : Flow → str`

  Receives the Flow object `self`, and returns the name of the source where the flow represented by `self` starts.

- `same_source : Flow × Flow → bool`

  Receives the Flow object `self` and another Flow object, and returns the value `True` or `False` indicating whether they both have the same source.

- `copy_flow : Flow → Flow`

  Receives the Flow object `self`, and returns a copy of itself.

- At least the standard Python methods `__str__` and `__eq__`.

  Furthermore, the method `__hash__`, which should enable to make the Flow class hashable.

Policy. The Policy ADT represents the current information flow policy that determines which flows are illegal. It should take into consideration all the patterns that are being considered.

- `__init__ : Policy →`

  Receives a new Policy object `self` and initializes it (it is automatically returned at instantiation time).

- `add_pattern : Policy × Pattern →`

  Receives the Policy object `self` and a Pattern object, and if `self` does not have any pattern with the same name, it changes itself to include that additional pattern.

- `delete_pattern : Policy × str →`

  Receives the Policy object `self` and a name as a string, and if `self` has a pattern with the same name, it changes itself to eliminate that pattern.

- `get_vulnerabilities : Policy → list(str)`

  Receives the Policy object `self`, and returns the list of vulnerability names that are valid in `self`.

- `get_vulnerabilities_source : Policy × str → list(str)`

  Receives the Policy object `self` and a source name, and returns the list of vulnerability names for which the given name is a source.

- `get_vulnerabilities_sanitizer : Policy × str → list(str)`

  Receives the Policy object `self` and a sanitizer name, and returns the list of vulnerability names for which the given name is a sanitizer.

- `get_vulnerabilities_sink : Policy × str → list(str)`

  Receives the Policy object `self` and a sink name, and returns the list of vulnerability names for which the given name is a sink.

- `get_sanitizers_vulnerability : Policy × str → list(str)`

  Receives the Policy object `self` and a vulnerability name, and returns the list of sanitizer names corresponding to the given vulnerability.

- `illegal_flows : Policy × Label × str → Label`

  Receives the Policy object `self`, a Label object and a name as a string, and returns a new Label object representing only the flows represented in the original Label object that should not reach a sink with the given name. Tip: It can be obtained by removing from the sources in a copy of an original label all those that are associated to patterns for which the name is not a sink.

- At least the standard Python method `__str__`, which should enable to print the Policy's patterns by alphabetical order of vulnerability name.

Label. The Label ADT is used to represent all the information flows that might have influenced a certain piece of data. It allows to collect information about the sources from which the information might have originated. For instance, the expression `src()+3*x` is influenced by a flow from name `src` and by any flows that might have influenced `x`. The Label ADT includes the following methods:

- `__init__ : Label × Policy →`

  Receives a new Label object `self` and a Policy object, and initializes it (it is automatically returned at instantiation time).

- `add_if_source : Label × str →`

  Receives the Label object `self` and a source name as a string, and changes `self` so as to include a new flow starting at that source for all vulnerabilities for which that name is a source.

- `clear_flows : Label × str →`

  Receives the Label object `self` and a vulnerability name as a string, and changes `self` so as to erase any flows associated to the given vulnerability.

- `sanitize : Label × str →`

  Receives the Label object `self` and a sanitizer name as a string, and changes `self` so as to erase all Flows associated to all vulnerabilities that have that sanitizer.

- `get_flows_vulnerability : Label × str → set(Flow)`

  Receives the Label object `self` and a vulnerability name, and returns the set of flows corresponding to the given vulnerability.

- `copy_label : Label × → Label`

  Receives the Label object `self` and returns a new Label object that is a deep copy of `self`.

- `label_combine : Label × Label → Label`

  Receives the Label object `self` and another Label object, and returns a new Label object that results from combining the two that were received. The new Label object should include all Flows for all vulnerabilities in both of them.

- At least the standard Python methods `__str__` and `__eq__`.

  Furthermore, the method `__add__`, which should enable to use `label_combine` by writing the operator '+' to combine two Label objects.

3.3 Analysis

For each method and function, include in the documentation of the code its complexity using the big-O notation. It should be expressed in terms of the number of Patterns (P), the maximum number of names each of them includes (N), and the number of nodes in the program slice's AST (S).

4 Specification of the tool - Checkpoint 2

The aim of this second stage is to finish setting up the basic abstract data types that your program will operate with, and to add functionality to the interaction cycle to help understand the structure of the AST that will be analysed in the third stage.

4.1 Explicit and Implicit Flows

There are two main ways in which information flows can be encoded into a program: explicitly, via assignments, and implicitly via the control flow, when a condition is tested and different control paths happen depending on the result.

Explicit flows. In the following example, there are explicit flows from `source` to `x` and from `x` to `y`:

```
x = source()
...
y = x
...
sink(y)
```

Implicit flows. In the following example, there are implicit flows from `x` to `y` in both branches of the conditional, as the value of `x` is influencing the value of `y`:

```
x = source()
...
if (x%2 == 0):
   y = 0
else:
   y = 1
...
sink(y)
```

A similar example can be written using while loops. While in this example very little information is leaked from `x` to `y`, it is possible to use brute-force to leak the entire information contained in a variable solely by means of implicit flows.

Implicit flows can be quite subtle and take complex paths. However the concept can be simplified as so:

When a conditional (`if`) or loop (`while`) occurs, information can flow from all variables that occur in the condition to all variables that are assigned within the branches or body that depend on the condition.

This means, in particular, that when there are nested loops and conditionals, an assignment that occurs in an inner block is potentially affected by all the variables that appear in the conditions of all the outer tests on which that block depends, i.e. that form the *context* in which the

assignment is taking place. It is then helpful to set up a data structure that keeps track of the implicit flows that might affect the inner blocks of conditionals and loops, as the control flow enters and exits nested blocks. This will be done by the ADT Context, as defined bellow.

4.2 Commands

The following command should be added to the interaction cycle of the tool:

`j`	pretty print the AST represented in JSON

Command `j`. The tool should print the AST of the currently stored program slice, using the JSON format, and indentations that help visualize its structure. Since the AST contains information tht is not relevant to the analysis, only the following set of object keys should be included: `args`, `func`, `id`, `left`, `right`, `comparators`, `ast_type`, `targets`, `value`, `test`, `body`, `orelse`.

If there is an error in the process, the program should raise an exception with an appropriate error message. You can however assume that the types of the arguments are correctly introduced by the user (so you do not need to validate the types of the arguments).

4.3 Abstract Data Types

Your tool should include and use an implementation of the abstract data types Pattern, Flow, Policy and Label with the interface defined below.

1. To make sure that your program respects the abstraction barriers of the ADTs, **all field attributes of these classes should be hidden**, i.e., should be named using the double underscore: `__field_name`.

2. Document all of the classes and methods in order to make explicit: the **internal representation** of the objects of each class, as well as the **parameter and return types** of each method.

Context. The Context ADT represents the information that is carried by the control flow of the program being analysed. It will be used to keep track of the labels associated to the conditionals on which the a block depends, and which represent the implicit flows that can be carried into the different dependent blocks of the program slice that is analysed. It includes the following methods:

- `__init__ : Context →`

  Receives a new Context object `self`, and initializes it (it is automatically returned at instantiation time).

- `enter_block_label : Context × Label →`

  Receives the Context object `self` and a Label object that represents the implicit flows that are created when entering a new block, and updates the Context so as to represent the updated context label.

- `exit_block : Context →`

  Receives the Context object `self`, and updates the Context so as to discard the implicit flows that were added when entering the innermost block.

- `in_block : Context →`

  Receives the Context object `self`, and returns the value `True` or `False` indicating whether `self` holds a label corresponding to some block.

- `get_block_label : Context →`

  Receives the Context object `self`, and returns the Label object that represents all the implicit flows that affect the current block.

- At least the standard Python method `__str__`.

LabelMap. The LabelMap ADT represents a mapping from variable names to labels. It will be used to keep track of the information flows that might have affected the current values that are held in the variables of the program slice. It includes the following methods:

- `__init__ : LabelMap →`

  Receives a new LabelMap object `self`, and initializes it (it is automatically returned at instantiation time).

- `is_labelled : LabelMap × str →`

  Receives the LabelMap object `self` and a name as a string, and returns the value `True` or `False` indicating whether the name is given a label in `self`.

- `map_name_to_label : LabelMap × str × Label →`

  Receives the LabelMap object `self`, a name as a string and a Label object, and updates `self` so as to map the given name to the given label.

- `labmap_combine : LabelMap × LabelMap → Label`

  Receives the LabelMap object `self` and another LabelMap object, and returns a new LabelMap object that results from combining the two that were received. The new LabelMap object should include all variable names in the domain of both, and map them to all Flows for all vulnerabilities that were associated to them in both.

- `get_copy_label : LabelMap × str → Label`

  Receives the LabelMap object `self` and a name as a string, and returns a new Label object that is a (deep) copy of the label that is mapped by `self` from the given name.

- At least the standard Python methods `__str__` and `__eq__`.

  Furthermore, the method `__add__`, which should enable to use `labmap_combine` by writing the operator '+' to combine two LabelMap objects.

IllegalFlows. The IllegalFlows ADT is used to collect all the illegal flows that are discovered during the analysis of the program slice. It includes the following methods:

- `__init__ : IllegalFlows × Policy →`

  Receives a new IllegalFlows object `self` and a Policy object, and initializes it (it is automatically returned at instantiation time).

- `get_illegal_flows : IllegalFlows × Label × str →`

  Receives the IllegalFlows object `self`, a Label object and a name as a string, and updates `self` so as to include any illegal flows that result when information with the given label reaches a sink with the given name.

- At least the standard Python method `__str__`, which should print the illegal flows for each vulnerability name, sorted by lexicographic[6] order of vulnerability name and of source name. The illegal flows should be presented as pairs `flow->sink`, for each individual flow of each vulnerability pattern, or the information of that no illegal flow was found for that vulnerability. For each vulnerability for which illegal flows were found, it should print the recommended sanitizers to prevent that vulnerability. Example:

  ```
  Vulnerability SQL injection A detected!
  Illegal flows: get->execute, get_object_or_404->execute,
  QueryDict->execute
  Recommended sanitizers: ['mogrify', 'escape_string']
  Vulnerability XSS not detected.
  ```

  (The precise format is not important, just the content that is included in the output and its structure.)

4.4 Analysis

For each method and function, include in the documentation of the code its complexity using the big-O notation. It should be expressed in terms of the number of Patterns (P), the maximum number of names each of them includes (N), and the number of nodes in the program slice's AST (S).

[6] Similar to alphabetic order, but includes numbers.

5 Specification of the tool – Checkpoint 3

The aim of the third and last stage is to implement the analysis of the program slices' AST. Subsection 5.1 presents the structure of the ASTs.

5.1 The AST type

An AST is a Python dictionary that represents a statement node of an Abstract Syntax Tree. It contains a key `"ast_type"`, which is mapped to a string that determines the type of that node, refered to as *root*, and determines the structure of the node. We will consider the following types:

- Type `Module`.

  Represents a program slice. Contains a key `"body"`, that is mapped to a list of statement nodes:

```
{
  "ast_type": "Module",
  "body": [
    ...
  ]
}
```

- Type `Assign`.

  Represents an assignment statement. Contains a key `"targets"`, that is mapped to a list which contains a node of type `Name`, and a key `"value"`, that is mapped to an expression node:

```
{
  "ast_type": "Assign",
  "targets": [
    {
      "ast_type": "Name",
      ...
    }
  ],
  "value": {
    ...
  }
  ...
}
```

- Type `Expr`.

  Represents an epression statement. Contains a key `"value"`, that is mapped to an expression node:

```
{
  "ast_type": "Expr",
  "value": {
    ...
  }
}
```

- Type `If`.

  Represents an if statement. Contains a key `"test"`, that is mapped to an expression node, and keys `"body"` and `"orelse"` that are mapped to lists of ASTs:

```
{
  "ast_type": "If",
  "body": [
    ...
  ],
  "orelse": [
    ...
  ],
  "test": {
    ...
  }
}
```

- Type `While`.

  Represents a while statement. Contains a key `"test"`, that is mapped to an expression node, and keys `"body"` and `"orelse"` that are mapped to lists of ASTs:

```
{
  "ast_type": "While",
  "body": [
    ...
  ],
  "orelse": [
    ...
  ],
  "test": {
```

```
    ...
  }
}
```

Expression nodes include the following:

- Type `Constant`.

  Represents a constant expression. Contains a key `"value"`, that is mapped to a value (integer,boolean, string):

  ```
  {
    "ast_type": "Constant",
    "value": true
  }
  ```

- Type `Name`.

  Represents a name expression. Contains a key `"id"`, that is mapped to a string representing a variable name:

  ```
  {
    "ast_type": "Name",
    "id": "a",
  }
  ```

- Type `BinOp`.

  Represents a binary operation expression. Contains keys `"left"` and `"right"`, that are mapped to expression nodes corresponding to the arguments of the operation:

  ```
  {
    "ast_type": "BinOp",
    "left": {
      ...
    },
    "right": {
      ...
    }
  }
  ```

- Type `Compare`.

  Represents a comparison. Contains a keys `"left"`, that is mapped to an expression node, and a key `"Comparators"`, that is mapped to a list containing an expression node, both of which correspond to the arguments of the comparison:

```
{
  "ast_type": "Compare",
  "comparators": [
    {
      ...
    }
  ],
  "left": {
    ...
  }
}
```

- Type `Call`.

  Represents a function call expression. Contains a key `"func"`, that is mapped to a dictionary which in turn contains the key `"id"` that is mapped to a string representing a function name, and a key `"args"`, that is mapped to a list of expression nodes corresponding to the arguments:

```
{
  "args": [
    ...
  ],
  "ast_type": "Call",
  "func": {
    "ast_type": "Name",
    "id": "fun",
  }
}
```

5.2 Commands

The following command should be added to the interaction cycle of the tool:

`a`	analyse the current program slice

Command `a`**.** The tool should perform an analysis on the AST of the currently stored program slice, and report on encoded vulnerabilities and illegal information flows. The illegal flows should be presented as pairs `flow->sink`, organized and lexigraphically sorted by vulnerability name and by source name, or the information of that no illegal flow was found for that vulnerability. For each vulnerability for which illegal flows were found, it should print the recommended sanitizers to prevent that vulnerability.

If there is an error in the process, the program should raise an exception with an appropriate error message. You can however assume that the types of the arguments are correctly introduced by the user (so you do not need to validate the types of the arguments).

5.3 Abstract Data Types

Your tool should include and use an implementation of the abstract data types Pattern, Flow, Policy and Label with the interface defined below.

1. To make sure that your program respects the abstraction barriers of the ADTs, **all field attributes of these classes should be hidden**, i.e., should be named using the double underscore: `__field_name`.

2. Document all of the classes and methods in order to make explicit: the **internal representation** of the objects of each class, as well as the **parameter and return types** of each method.

Analyser. The Analyser ADT represents the analysis functionality of the tool. It includes methods for traversing different program constructs and returning the information collected during the traversal. It includes the following methods:

- `__init__ :` $\texttt{Analyser} \times \texttt{Policy} \times \texttt{IllegalFlows} \rightarrow$

  Receives a new Analyser object `self`, a policy object and an IllegalFlows object, and initializes it (it is automatically returned at instantiation time).

- `expr_name :` $\texttt{Analyser} \times \texttt{AST} \times \texttt{LabelMap} \rightarrow \texttt{Label}$

  Receives a new Analyser object `self`, an AST expression node of the type Name, and a LabelMap object, and returns a new Label object representing the information flows that are carried by the name. The label should be the one mapped to the name by the given labelmap, or a fresh label if it is not in its domain.

- `expr_binop :` $\texttt{Analyser} \times \texttt{AST} \times \texttt{LabelMap} \rightarrow \texttt{Label}$

  Receives a new Analyser object `self`, an AST expression node of the type BinOp, and a LabelMap object, and returns a new Label object representing the information flows that are carried by the name. The label should contain a combination of the flows carried by each of the sub-expressions of the binary operation. (Tip: You can call `expr_call` by mutual recursion.)

- `expr_compare :` $\texttt{Analyser} \times \texttt{AST} \times \texttt{LabelMap} \rightarrow \texttt{Label}$

  Receives a new Analyser object `self`, an AST expression node of the type Compare, and a LabelMap object, and returns a new Label object representing the information flows that are carried by the name. The label should contain a combination of the flows carried by each of the sub-expressions of the comparison. (Tip: You can call `expr_call` by mutual recursion.)

- `expr_call : Analyser × AST × LabelMap → Label`

  Receives a new Analyser object `self`, an AST expression node of the type Call, and a LabelMap object, and returns a new Label object representing the information flows that are carried by the function call. The flows should include:

  - those produced by each of the arguments of the function call; (Tips: You can call `expr_label` by mutual recursion. At first you can assume that there is only one argument)
  - the function name itself if it is a source;
  - but, if the function name is a sanitizer, not those sanitized by it.

  Furtermore, since the function name can be a sink, if any of the above flows, *or those stored in the program Context*, would encode an illegal flow when reaching the function name, they should be collected (to be reported at the end of the analysis).

- `expr_label : Analyser × AST × LabelMap → Label`

  Receives a new Analyser object `self`, an AST expression node, and a LabelMap object, and returns a new Label object representing the information flows that are carried by the expression. (Tip: You can call `expr_name`, `expr_binop`, `expr_compare` and `expr_call`; don't forget the case of the Constant expression.)

- `traverse_assign : Analyser × AST × LabelMap → LabelMap`

  Receives a new Analyser object `self`, an AST statement node of the type Assign, and a LabelMap object, and returns a new LabelMap object that is like the one received as argument, but that also takes note of the flows introduced by the assignment. This means that the flows in the label of the variable being assigned should only include:

  - those carried by the expression that is being assigned to the variable;
  - *those carried by the current program Context, if any*.

- `traverse_if : Analyser × AST × LabelMap → LabelMap`

  Receives a new Analyser object `self`, an AST statement node of the type If, and a LabelMap object, and returns a new LabelMap object that is like the one received as argument, but that also takes note of the flows introduced by the if condition. This means that the labels in the returned labelmap should include the combination of the labels that would be obtained by analysing the two branches of the conditional *independently*. (Tip: You can call `traverse` by mutual recursion.)

  In order to take into consideration the implicit information flows generated by the if condition, before analysing the branches, the label of the test should be added to the Context, and should afterwards be removed.

- `traverse_while : Analyser × AST × LabelMap → LabelMap`

  Receives a new Analyser object `self`, an AST statement node of the type While, and a LabelMap object, and returns a new LabelMap object that is like the one received as argument, but that also takes note of the flows introduced by the while loop. This means that the labels in the returned labelmap should include the combination of the labels that would be obtained by analysing the body a different number of times, *independently*. (Tips: You can call `traverse` by mutual recursion. You can start by considering that the loop can only execute zero or one times.)

  In order to take into consideration the implicit information flows generated by the while condition, before analysing the loop, the label of the test should be added to the Context, and should afterwards be removed.

- `traverse : Analyser × AST × LabelMap → LabelMap`

  Receives a new Analyser object `self`, an AST statement node, and a LabelMap object, and returns a new LabelMap object that is like the one received as argument, but that also takes note of the flows introduced by the statement. (Tip: You can call `traverse-assign`, `traverse-if` and `traverse-while`, as well as `expr_label` for the Expr case.)

(*) In the above specification of the methods, the functionality related to the treatment of implicit flows is marked with '*'. This component is more advanced; you can start by implementing the methods without it, and add it later when all else is working.

6 Development, submission and evaluation

Important Dates The development of the Project will be structured as follows:

Checkpoint 1 The interaction cycle, and basic ADTs for storing and using vulnerability pattern and the policy that they represent. **Deadline 27 March**.

Checkpoint 2 More advanced ADTs to supporting tracking of information flows, recognition and reporting of vulnerabilities during the analysis. **Deadline 12 April**.

Checkpoint 3 Main functions for traversing the AST and extracting the vulnerabilities, optional upgrade of the reprersentation of vulnerabilities. **Deadline 24 April**.

Discussions About the project developed by each group. Will take place during the week **25 to 29 of April**.

Authorship. Projects are to be solved in groups of 1 or 2 elements. Both members of the group are expected to be equally involved in solving, writing and presenting the project, and share full responsibility for all aspects of all components of the evaluation. You can discuss with your colleagues, but each group is expected to write all of its own code. Presence at the discussion and tool demonstration is mandatory in order to have a grade.

Testing. We provide program slices and patterns to assist in testing your code. It is however your responsibility to perform more extensive testing to ensure the correctness and robustness of the tool. Note that for the purpose of developing and testing the tool, you do *not* need to have knowledge about the names of vulnerabilities, sources, sanitizers and sinks of real world vulnerabilities. You can use vulnerability patterns with any function names, as this will not affect the ability of the tool to manage meaningful patterns.

Support. During both theoretical and practical classes you will be offered guidance for the development of the project. On the course's web-page there will tips, as well as examples of valid inputs and outputs that you can use to help test your solution. If you need any help, do not hesitate to ask. You can also you the Slack channel `#project` for questions and discussions about the project.

Project submission. At each checkpoint, the project should be submitted in Fenix as a zip file containing only .py files with the implementation of the tool.

Evaluation. The weight of each stage of the project is **20%, 35% and 45%**, respectively, and will be evaluated according to:

1. How well it meets the specification;
2. The efficiency of the implementation;
3. The analysis;
4. The discussion. The baseline grade for the group will be obtained using the above criteria. During the discussion, both students in the groups are expected to be able to demonstrate knowledge of all details of these two components in order to be graded for the project.

Good work!