Custom Neutralizations

Introduction

Tainted Flow Analysis

The root cause of many security breaches is trusting unvalidated input. This could be:

- Input from the user, which could be considered as **tainted** (possibly controlled by an adversary), i.e user is considered as an untrusted source
- Data, assuming it is **untainted** (must not be controlled by an adversary), i.e. sensitive data sinks rely on trusted (untainted) data

> ⓘ **Source** locations are those code places from where data comes in, that can be potentially controlled by the user (or the environment) and must consequently be presumably considered as tainted (it may be used to build injection attacks).
>
> **Sink** locations are those code places where consumed data must not be tainted.

The goal of **Tainted Flow Analysis** is to detect tainted data flows:

Prove, for all possible sinks, that tainted data will never be used where untainted data is expected.

> ⓘ Kiuwan implements Tainted Flow Analysis by inferring flows in the source code of your application:
>
> - What sinks are reached by what sources
> - If any flows are illegal, i.e., whether a tainted source may flow to an untainted sink without going across a sanitizer

When inferring flows from an untainted sink to a tainted source, Kiuwan can detect if any well-known *sanitizer* is used, dropping those flows and thus avoiding to raise false vulnerabilities.

Kiuwan contains a built-in library of sanitizers for every supported programming language and framework.

These sanitizers are commonly used directly by programmers or by frameworks. And Kiuwan detects their use.

Read more here Understanding Data-Flow Vulnerabilities

Data Neutralization Model

Complex *subsystems* that accept string data that may hold commands or instructions need neutralization of inputs targeted to them.

If untrusted input entering the subsystem may result in unexpected execution of commands/actions, an injection security flaw exists. Examples of such subsystems that are candidates for injection attacks are:

- The operating system command interpreter
- Data repository with the SQL engine

- XML parser
- XPath / XQuery evaluator
- LDAP directory service API
- Script engines
- Regexp compilers (e.g. the pcre_replace() PHP function with /e pattern modifier)

> ⓘ The root cause of most web security flaws:
>
> - Too much trust in external input (but HTTP request msg could be change ad-libitum by the hacker): headers (incl. cookies), request URL, body (incl. hidden fields).
> - No adequate input validation / output sanitization / canonicalization – normalization.

The first defense line against application attacks is adequate **input validation**.

- Should be **positive**, "accept only which is known to be good" (**whitelist**), not negative, "reject what is known to be bad" (blacklist).
- Sometimes **output escaping** is a good thing (e.g. against XSS; but less against SQLi and other attacks)

> ⓘ Good practice says: "**filter on input, escape on output**".
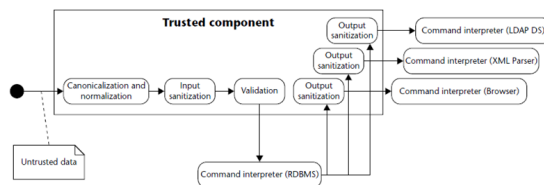
- **Canonicalization / Normalization**
  - *Canonicalization* is the process of lossless reduction of input to its equivalent simplest known form (for example, replacing .. and . in a pathname to produce canonicalized pathname, Unicode canonical equivalence…).
  - *Normalization* is the process of lossy conversión of input data to the simplest form (e.g. converting a text input into one value from a fixed set, removing accents, removing whitespace, stop words and punctuation chars, lower-/upper-casing…).
- **Sanitization**
  - Ensuring that data conforms to the requirements of the subsystem to which it is passed, including security requirements related to data leakage or sensitive data exposure across the trust boundary. This may include the removal of unwanted characters, escaping metacharacters, etc.
- **Validation**
  - Ensuring that input falls within the expected domain of valid program input: type /numeric range requirements, input invariants…

Kiuwan contains a built-in library of sanitizers for every supported programming language and framework. These sanitizers are commonly used directly by programmers or by frameworks. And Kiuwan detects their usage.

But if you are using your sanitizers, Kiuwan could not recognize them as such, detecting false "tainted data flow". In this case, you should let Kiuwan be aware of them.

The goal of this section is **to teach you how to incorporate custom sanitizers to the Kiuwan built-in library**.

During the next section, we will use the terms "sanitizers" and "neutralization routines" as synonyms.

Neutralization Routines (a.k.a Sanitizers)

> ⓘ A **Neutralization Routine** (or **Sanitizer**) is understood as any piece of code that can assure that any tainted data got as input produces untainted as output.

This documentation is not related to how to build custom neutralization routines, but how to add your own custom neutralization routines to Kiuwan.

The process consists of:

- First, **let Kiuwan know your routine**
    - Depending on the programming language you are analyzing, the so-called "routine" can be a function, a method of a class, etc.
- Second, **let Kiuwan know that it's a neutralization**
    - Kiuwan provides some ways to define your routine (we will see it later) but, regardless of it, you need to indicate that routine as "neutralization".

Next, for instruction purposes, we will follow these steps using Java as the programming language. Differences with other programming languages will be further detailed.

Specifying custom neutralization routines

> ⓘ Any custom neutralization routine must be defined in a **custom neutralization file** (XML format).
>
> The name of the file is irrelevant but the location is quite important.

Locations and precedence

Neutralization routines can be configured at different **scopes**

- *Single-analysis,*
- *Application-specific* and
- *System-wide.*

Depending on the location of the XML file, precedence and scope will change.

Precedence and scope of configurations are as follows:

- **Single-Analysis**
    - Neutralizations can apply only to a unique analysis.
    - In this case, the XML file should be located at:
        - `[analysis_base_dir]/libraries/[technology]`

- **Application-specific**
    - Neutralizations can apply to all analyses of a specific application.
    - In this case, the XML file should be located at:
        - `[agent_home_dir]/conf/apps/[app_name]/libraries/[technology]`

- **System-wide**
    - Neutralizations can apply to all analyses of all applications.
    - In this case, the XML file should be located at:
        - `[agent_home_dir]/conf/libraries/[technology]`

    - Exceptions to this rule are:
        - CPP engine reads from …/libraries/c
        - objective engine reads from …/libraries/objetivec and …/libraries/c

> ⓘ **Legend**
>
> - [*agent_home_dir*]: local installation directory of Kiuwan Local Analyzer (KLA)
> - [*analysis_base_dir*]: the root directory of application source code to be analyzed, as specified by "-s" option of KLA CLI (Command Line Interface), or in "Folder to analyze" input box when using KLA GUI (Graphical User Interface)
> - [*app_name*]: name of the app to be analyzed, as specified by "-n" option of KLA CLI (Command Line Interface), or in "Application name" input box when using KLA GUI (Graphical User Interface)
> - [*technology*] : name of the Kiuwan technology, as specified in [agent_home_dir]/conf/LanguageInfo.properties

> ⚠ **Be careful**
>
> Never save custom libraries files or edit existing files in folder `[agent_home_dir]/libraries/{tech}`, because this folder is going to be removed when the engine is updated.

As a general recommendation, we suggest naming the XML file as *[technology]_custom_neutralizations. xml* (this will help to identify your custom files from Kiuwan's own files).

Therefore, the next sections will use **java_custom_neutralizations.xml** as the name for our custom file.

Creating a custom "Library" of neutralization routines

You don't need to create an XML file for every single neutralization routine.

Instead, you will include all of them in a single file identified as a library of custom neutralization routines, with a name for it.

Library identification will be an XML element such as:

```
<library name="java.custom.libraries"/>
```

As a suggestion, we recommend using something like: *[technology].custom.library*

Please refer to the schema file for editing custom libraries of neutralization routines.

As said above, a Neutralization Routine is a piece of code that assures that any tainted data got as input produces untainted data as output.

That piece of code is typically a function or a class method (depending on whether your technology is object-oriented or not).

> (i)   Then, what you must do in the XML file is to properly **declare** such routine and **mark** it as a
>   neutralization routine.

To declare the routine, you must include the element. The schema file describes the allowed set of elements that form part of the library.

For our purposes, commonly used elements are either **class** or **function**, depending on the language.

Once, the routine is declared, it must be marked as a neutralization routine as follows. See the reference section for more details on how to declare a routine.

Examples

Let's see some explained examples of custom neutralizations:

Example 1 (Java)

In this example the method *validate* is a custom neutralization for a path from a source to a path traversal sink. The input of method *validate* is neutralized and the output, (referred by argpos -1 in the neutralization definition in the XML library), is untainted after the validation is executed.

The next source code shows an example of how to use the neutralization:

```
package com.mycompany.onepackage;

import com.mycompany.otherpackage.MyUtils;
import javax.servlet.http.HttpServletRequest ;
import java.io.FileInputStream;
public class MyClass {
    // ...
    public void methodThatAccessToFileSystem(HttpServletRequest req) {
        String inputFile = req.getParameter("file"); //inputFile tainted
        inputFile = MyUtils.validate(inputFile + ".tmp"); //inputFile
untainted after validation
        return new FileInputStream(SAFE_DIR.getAbsoluteFile() + inputFile);
    }
    // ...
}

=======================================

package com.mycompany.otherpackage;

import com.mycompany.IMyUtilsClass;

public class MyUtils implements IMyUtilsClass {
        // ....
        public String validate(String value) {
                // ...
        // perform string value validation/Canonicalization/Normalization
/Sanitization
        // ...
                return value; // once cleaned up
    }
}
```

And this is how you should declare the neutralization method in the library XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="http://www.optimyth.com/schema/definitions
/library_metadata"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        name="java.custom.libraries" standard="custom" technology="java">
  <class name="com.mycompany.otherpackage.MyUtils" kind="class"
supertypes="com.mycompany.IMyUtilsClass">
    <method name="validate" signature="validate(java.lang.String)" match="
name">
      <return type="java.lang.String"/>
      <neutralization argpos="-1" kind="path_traversal" resource="web" />
    </method>
  </class>
</library>
```

ⓘ **Do not forget**

- types have to be fully qualified
- specify return type if the method has one
- no need to declare parameters names in the method signature, just the fully qualified
  types

Neutralization **argpos**, **kind** and **resource** arguments will be discussed later...

Example 2 (Java)

In the next example the neutralization only affects to filesystem resources:

```
package com.mycompany.onepackage;

import com.mycompany.otherpackage.CustomFile;
import javax.servlet.http.HttpServletRequest;
import java.io.FileInputStream;
public class MyClass {
  // ...
  public void methodThatAccessToFileSystem(HttpServletRequest req) {
    String inputFile = req.getParameter("file"); //inputFile tainted
    CustomFile file = new CustomFile(inputFile);
    file.sanitize(); //file untainted after sanitization
    return new FileInputStream(SAFE_DIR.getAbsoluteFile() + file);
  }
}

=====================================

package com.mycompany.otherpackage;

import java.io.File;

public class CustomFile extends File {
        //..
        public void sanitize() {
                // perform file sanitization
        }
}
```

Neutralization declaration in the library XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="http://www.optimyth.com/schema/definitions
/library_metadata"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         name="java.custom.libraries" standard="custom" technology="java">
  <class name="com.mycompany.otherpackage.CustomFile" kind="class"
supertypes="java.io.File">
   <method name="sanitize" signature="sanitize()">
      <neutralization argpos="-2" kind="string" resource="filesystem"/>
    </method>
  </class>
</library>
```

Example 3 (Java)

In this example, a java annotation is used as neutralization.

```
package com.mycompany.onepackage;

import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.Connection;

public class Main {

        private Connection con;

        public static void main(String[] args) {
                String name = args[0];
                Item item = new Item(name);
            process(item);
        }
        public static void process(Item item) {
            try {
                    ResultSet rs = null;
                    Statement stmt = con.createStatement();
                    String input = item.getName();
                            rs=stmt.executeQuery("select * from items where
item_name="+ input);         // input is tainted
            } catch (...) {
            // ...
                }
            }
}

=====================================

package com.mycompany.onepackage;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
        // perform neutralization
}

=====================================

package com.mycompany.onepackage;

public class Item {
  String name;

  public Item(String name) {
    super();
    this.name = name;
  }

  @MyAnnotation()
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

So, you should declare the annotation as neutralization in the library XML file as:

```
<?xml version="1.0" encoding="UTF-8"?>
<library xmlns="http://www.optimyth.com/schema/definitions
/library_metadata"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         name="java.custom.libraries" standard="custom" technology="java">
  <annotation name="com.mycompany.onepackage.MyAnnotation">
      <neutralization kind="sql_injection" />
  </annotation>
</library>
```

Neutralization elements

A **neutralization** identifies an element (essentially a method call) as a vulnerability neutralizer, it is defined in Kiuwan by the following element:

```
<!ELEMENT neutralization (#PCDATA)*>
<!ATTLIST neutralization
  argpos CDATA #REQUIRED
  kind CDATA #IMPLIED
  resource %resource; #IMPLIED
>
```

- ### *argpos*

  the *argpos* attribute specifies what object (or objects) are *untainted* by the routine. It indicates which element is being neutralized by this neutralization. Depending on how your custom neutralization routine works, you should code a different value in this argument. Allowed values are:

Allowed values	Neutralization in XML

0..n: A non-negative value indicates that the argument at the given index (starting at 0)is being neutralized.
 	
    ○ *obj.call (arg1, arg2)*
        ■ *arg1* is neutralized when *argpos ="0"*
        ■ *arg2* is neutralized when *argpos ="1"*
        ■ Both are neutralized when *argpos ="0,1"*
	`<method name="call" signature="call(fqcn.Arg1Type, fqcn.Arg2Type)">`
	`    <neutralization argpos="0" kind="..." resource="..." />`
	`</method>`

-1: Target object (returned value) is being neutralized.
 	
    ○ *value = obj. call(arg1)*
        ■ *value* is neutralized when *argpos ="-1"*
	`<method name="call" signature="call(fqcn.Arg1Type)">`
	`    <return type="fqcn.ValueType"/>`
	`    <neutralization argpos="-1" kind="..." resource="..." />`
	`</method>`

|	

-2: Called object is being neutralized.
 	
     ○ *obj.call()*
        ■ *obj* is neutralized when *argpos ="-2"*
	``xml
	<method name="call" signature="call()">
	    <neutralization argpos="-2" kind="..."
	resource="..." />
	</method>
	``

	Neutralization routines could be defined in the same class where they are used, or in a different one, where you can invoke them through an object instantiation call or by a static call. Any combination of this and the *argpos* attribute values is possible.

- ### *kind*

  A neutralization routine is usually applied to a specific vulnerability type (or *kind*). The *kind* attribute indicates the type of vulnerability affected by this neutralization, like *XSS*, *sql_injection*, *open_redirect*, etc. Use *string* for general purpose neutralizations.

  You can include as many neutralization elements as vulnerability types your routine neutralizes. To see the exact attribute value, locate the vulnerability you need to neutralize, open the sink data and see the **Category** value.

  ```xml
  <neutralization argpos="-1" kind="sql_injection"/>
  ```

  ```xml
  <neutralization argpos="-1" kind="xss"/>
  ```

  If you want it, the neutralization applies to ALL the vulnerabilities (i.e. it's not specific to any vulnerability): set *string* as the value for the *kind* attribute
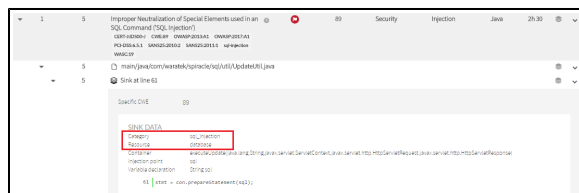
- ### *resource*

  A neutralization routine can also be specifically suited to a particular *resource* type.

  For example, your neutralization routine could be applied to database or filesystem resource types.

  Valid values of *resource* can be one of (`memory` `|os` `|configuration` `|environment` `|filesystem` `|formatstr` `|database` `|web` `|network` `|gui` `|crypto` `|other`).

  As above, check the Sink Data to set the appropriate value. That's the value you must indicate in a *kind* attribute.

Reference

Any **Custom Neutralization File (CNF)** must be an XML file with the following structure:

1. Reference to schema file
2. Definition of the custom Library of Neutralization routines
3. List of custom Neutralization routines

The schema file that may be used while editing custom neutralization libraries can be found in *[agent_home_dir]/lib.engine/analyzer.jar/resources/***library_metadata.xsd** file.

ⓘ

> **⊙ Be careful**
>
> For reference purposes only, you can check KiuwanLocalAnalyzer predefined library definitions in `[agent_home_dir]/libraries/{tech}`
>
> Never save custom libraries files or edit existing files in `[agent_home_dir]/libraries/{tech}`, because this folder is going to be removed when the engine is updated.