# Understanding Data-Flow Vulnerabilities

This section explains in detail how you can undestand a vulnerabilty as reported by Kiuwan.

**Contents**:



The explanation is focused on injection-related vulnerabilities, as an example of complex vulnerabilities.

We will first provide an **overview of Tainted Flow Analysis** (the theoretical basis behind the scenes), and then we will focus on **Kiuwan vulnerability reporting**.

## 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

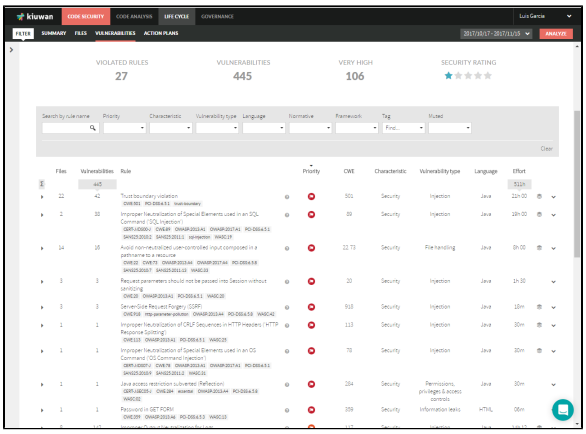
When inferring flows from an untainted sink to a tainted source, Kiuwan can detect if any well-known *sanit izer* is used, dropping those flows and thus avoiding to raise false vulnerabilities.

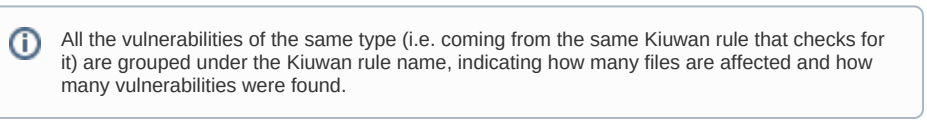
Kiuwan contains a built-in library of sanitizers for every supported programming language and framework.

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

## How to understand tainted-flow vulnerabilities

Vulnerabilities are reported under **Code Security > Vulnerabilities**.



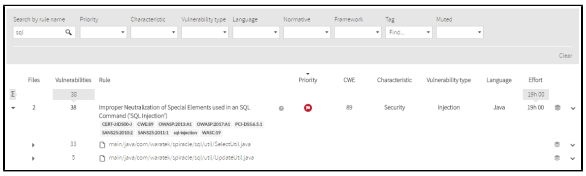


All the vulnerabilities of the same type (i.e. coming from the same Kiuwan rule that checks for it) are grouped under the Kiuwan rule name, indicating how many files are affected and how many vulnerabilities were found.

## Finding Vulnerabilities

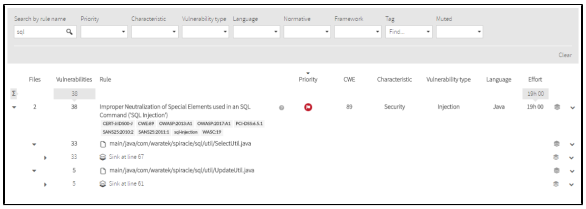
For our explanation, we will follow an example based on Waratek – Spiracle software, focusing on SQL injection vulnerabilities.

Search for the SQL injection vulnerabilities by entering "**SQL**" in the **Search by rule name** field.



In this image, we can see that 2 files are affected: there is a sink that is being fed tainted data. An **injection point**.

For every file there are a number of vulnerabillities:

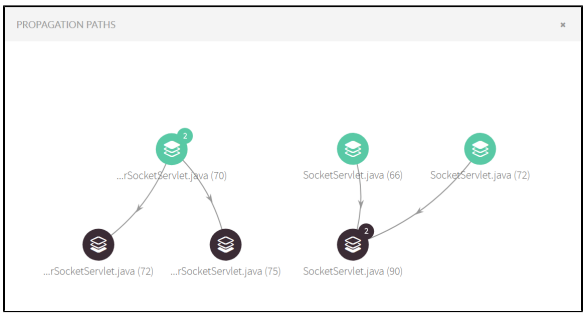


## Graphical view of sources and sinks

Every vulnerability offers the possibility to view a graph view of all the propagation paths for this item.

Click the  icon on the right:

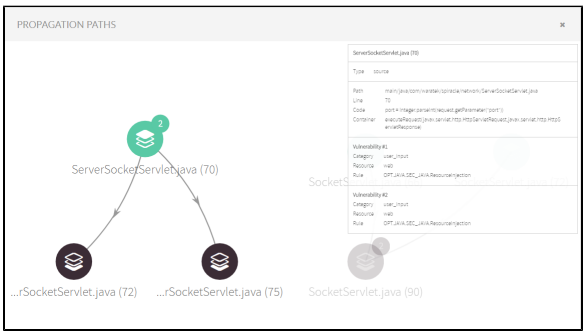
The following graph will open:




Tainted data flows are represented as **directed graphs** from sources (at the top) to sinks (at the bottom).

Any element may have a number that indicates in how many tainted data flows it participates.

You can see the detail of any element (source, sink or propagation node) by hovering the mouse overt it. A dialog will be displayed as in the image below:



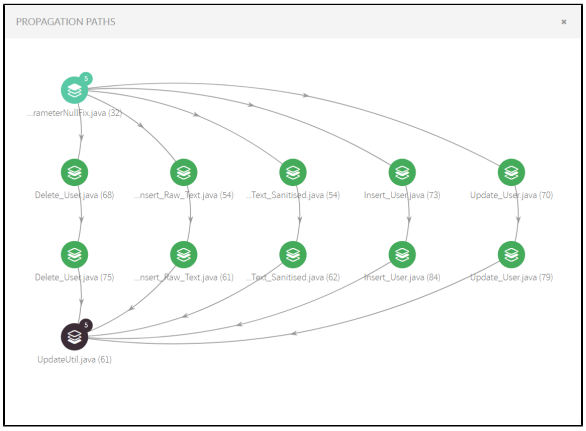

## Finding sources and sinks

> Clicking on the affected file will open all its affected sinks.

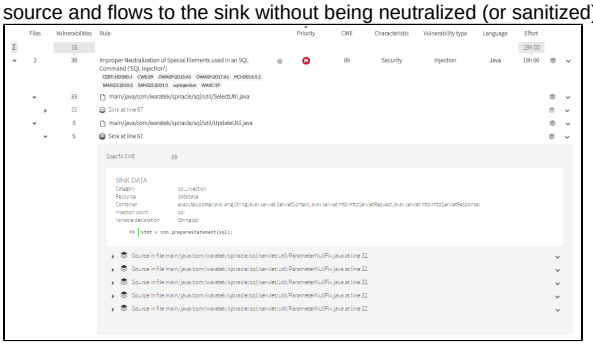
In this case, there's only one sink for every file (only one injection point), but it could be many. Kiuwan will display all the line numbers of the affected sinks.

If you want to see a **graphical view** of all the sources, sinks and tainted data flows for **a file** you can also click on the icon.

And the following graph will be displayed:



Clicking a sink displays its details as well as all the *sources* where data is collected from an untrusted source and flows to the sink without being neutralized (or sanitized).

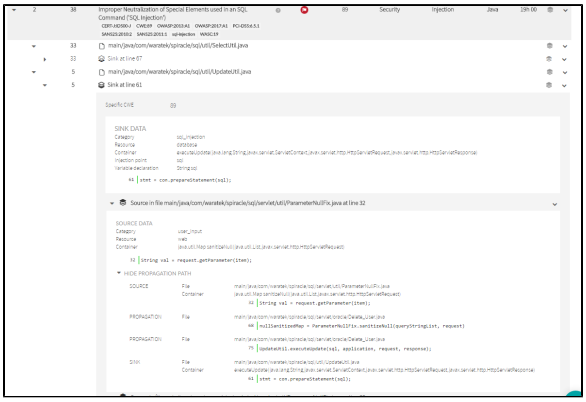


### Detailed information of a sink

> **Sink data**
>
> The sink details include the following information:
>
> - **Specific CWE**, related vulnerability as defined by CWE (hyperlink to definition at MITRE)
> - **Sink Data**
>   - **Category** (vulnerability type, e.g. sql_injection, xss, etc)
>   - **Resource** (affected resource, e.g. database, web, etc.)
>   - **Container** (function/method where the sink is located)
>   - **Injection point** (tainted data, i.e. variable name)
>   - **Variable declaration** (tainted variable declaration)
>   - **Source code line and text of sink**
>   - **List of Sources**
>     - Full list of sources with tainted-flow paths ending in the sink
>     - Every row indicates the source file and line where data coming from an untrusted source flows to the sink without being neutralized (or sanitized).

Most commonly, every source will be a different file. But depending on the flow path, you could find same source and line many times.

Let's see how to understand every source.



## Source's detailed information

Clicking on a source will open a frame with its detailed information.

**Source data**

Source detail includes following information:

- **Category** (source type, e.g. user_input, , etc)
- **Resource** (resource where the data is gathered, e.g. web, etc.)
- **Container** (function/method where the source is located)
- **Source code line and text of sink**
- **Propagation path** (data-flow between the source and sink)

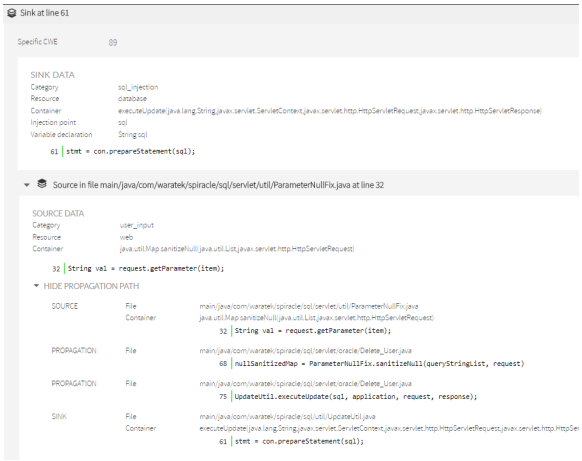
## Propagation Path

**Propagation Path**

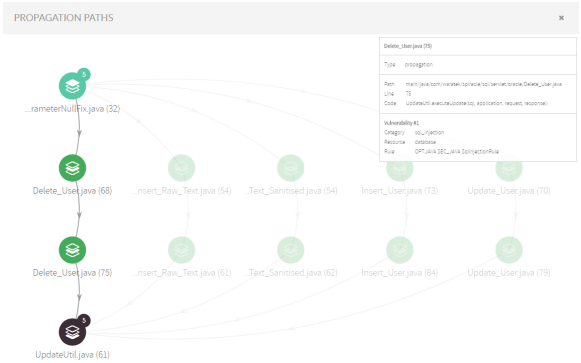
Important: You should not understand the **propagation path** as a typical stack of method calls. It's not that.

You should understand it as a **data-flow path**.

Let's look at the following example:




You can also view it in graphical mode:

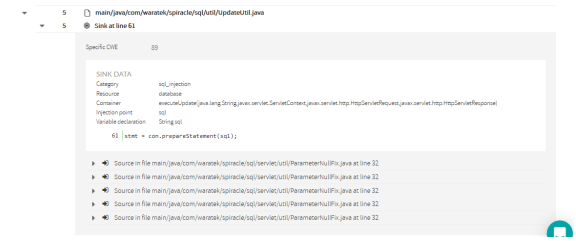


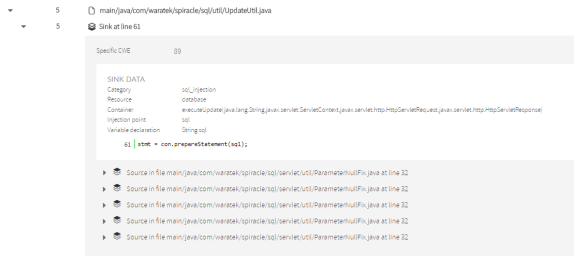

Any **propagation path** is composed of a **source node**, a **sink node** and as many **propagation nodes** as different methods are involved in the propagation path.

In the example, we can see the next propagation path (flowing from source to sink):

- The source node indicates that the file *ParameterNullFix.java* line 32 (within *sanitizeNull* method) gets the value from a request parameter.
    - This is marked a **source** because *HttpServletRequest.getParameter(..)* is considered as an untrusted input source (i.e. is directly manipulated by the user) and no neutralization routine has been found.
- The first propagation node indicates that *Delete_User.java* line 68 receives back the above tainted data.
- The second propagation node indicates that *Delete_User.java* line 75 sends the tainted data (via method parameter) to another object through *UpdateUtil.executeUpdate(…)* method call.
    - Looking at the source code, you could see that the tainted data is directly appended to a sql string without any neutralization, allowing this way to directly insert user code into the sql sentence.
- The sink node indicates that the file *UpdateUtil.java* line 61 (within *executeUpdate(…)* method) injects tainted data (sql sentence) to a PreparedStatement that finally is executed against the database.

Let's go back to the initial sink information.




As said above, for every sink there will appear the list of sources that are "feeding" that sink.

But you might be wondering why there are 5 sources with the same file name and line.

We've selected this specific example to show something that could happen in your own code. Let's explore the second source into detail.

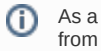

In the 2nd source you can see that the propagation nodes are different from the previous one.

While in the 1st the propagation it was through *Delete_User.java*, in the 2nd, it goes through a different file: *Insert_Raw_Text.java.*

Reminder sources for the same sink show that there are still other different propagation paths between the same source and the sink.

This example is a special case where the call sequence consists of 5 servlets calling a utility (*ParameterNullFix.sanitizeNull(..)*) to recover some request parameters, building an SQL sentence with this tainted data and sending the SQL sentence to another utility (*UpdateUtil.executeUpdate(…)* ) to execute the database update.

This is the reason Kiuwan shows one sink with 5 different sinks. That difference is because the propagation paths are through the different servlets. In this case, the easiest fix would be to sanitize the user data at the source, therefore remediating 5 found defects with only one fix.



As a summary, you can understand any injection vulnerability as a unique propagation path from source to sink, regardless of whether source and sink are the same.

## Data path

A vulnerability's data path information is complementary to its propagation path. Although they may similar, the data path adds detailed information on how the data flows through your code making it vulnerable.

Let us inspect the data path corresponding to the example shown in the Propagation path section:


A **data path** is composed of a series of steps that show how tainted data flows through the source code.

The data path shown in the example consists of (from source to sink):

- The container step shows the method where the user input is first detected in the analyzed source code. In this case, the *request* parameter contains user input data.
- The source step reports how the new *val* variable is initialized retrieving data from the *request* parameter, so Kiuwan considers it a tainted variable.
- In the next step the *val* variable is assigned to another object. In this case, the *outputMap* object is tainted because it contains input data that has not been sanitized.
- This map is then returned to the calling method, that is reported in the next step and belongs to a different file (*Delete_user.java*).
- The next step shows how the returned map is then inspected to get a key that has been tainted.
- The *name* variable is tainted and is injected into a query in the next step.
- The created *sql* string object is then sent to a method located in *UpdateUtil.java*.
- Once inside the method (which has been identified as the sink of the vulnerability), the tainted variable *sql* is used to construct the prepared statement that will be executed (and it contains data in which a user could have injected mailicious code).

## Configuration (parametersAsSources)

Some injection rules provide the ability to behave differently depending on a configuration parameter: **parametersAsSources**

This parameter makes the rule to consider the function/method parameters where the sink is contained as "*sources*". And sources are always being considered as "tainted".

If **parametersAsSources=true**, the rule performs a tainting path analysis to check if the parameters are neutralized since being received by the function/method until are consumed by the sink. If no neutralization is found, an injection vulnerability is raised. **By default**, **parametersAsSources=false.**

### Why should you change it to true ?

If your software being analyzed by Kiuwan is a complete application (i.e. it contains presentation plus logic and ddbb layers), you should let it be as false. In this way, Kiuwan will make a full tanting path analysis over the whole application code.

But, if your software is a "library", i.e. a software component that will be used by 3rd parties to build their own applications, **you should configure this property to true,** making Kiuwan perform that local tainting path analysis, thus guaranteeing that your library is protected against injection vulnerabilities regardless the usage by third parties.

As you can guess, setting to true and analyzing a complete application will result in a number of false positives.