Understanding Data-Flow Vulnerabilties

Goal of this section is to explain in some detail how you can understand a vulnerabilty as reported by Kiuwan.

We will focus the explanation on Injection-related vulnerabilities, as an example of complex vulnerabitlites.

To do it,

- firstly we will 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

Root cause of many security breaches is ***trusting unvalidated input***:

- Input from the user is considered as **tainted** (*possibly* controlled by adversary), i.e user is considered as a *untrusted* source
- Data is used, assuming it is **untainted** (*must not be* controlled by 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*:

For all possible sinks, prove 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”

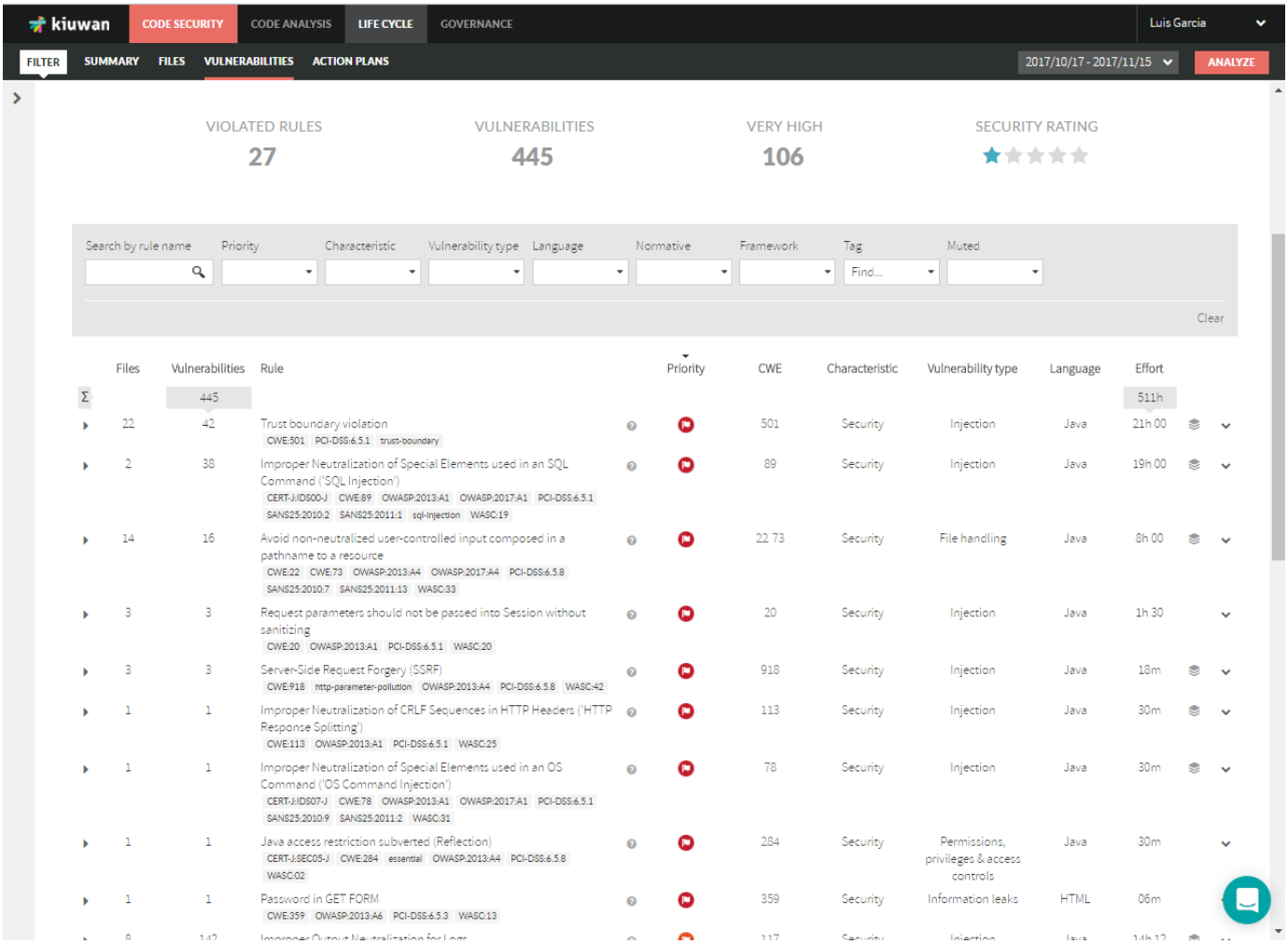
When inferring flows from an untainted sink to a tainted source, Kiuwan is able to detect if any well-known *sanitizer* is used, dropping those flows and thus avoiding to raise false vulnerabilities.

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 the use of them.

How to understand tainted-flow vulnerabilities

Vulnerabilities are reported at **Code Security – Vulnerabilities** section.

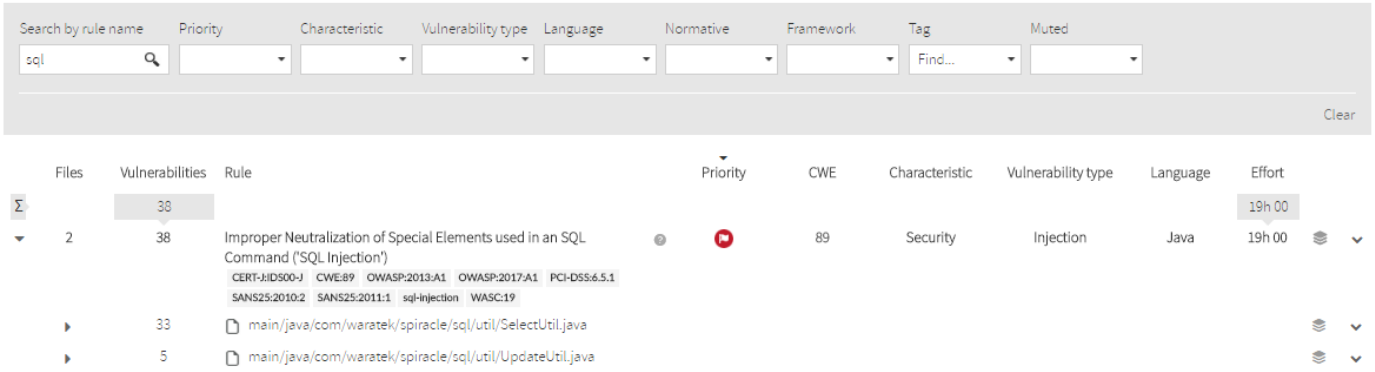


All the vulnerabilities of the same type (i.e. coming from the same Kiuwan rule that checks for it) are grouped under the Kiuwan rule name, indicating how many files are “affected” and how many vulnerabilities were found.

Finding Vulnerabilities (e.g. SQL Injection)

For our explanation, we will follow an example based on Waratek – Spiracle software, focused on SQL-Injection vulnerabilities.

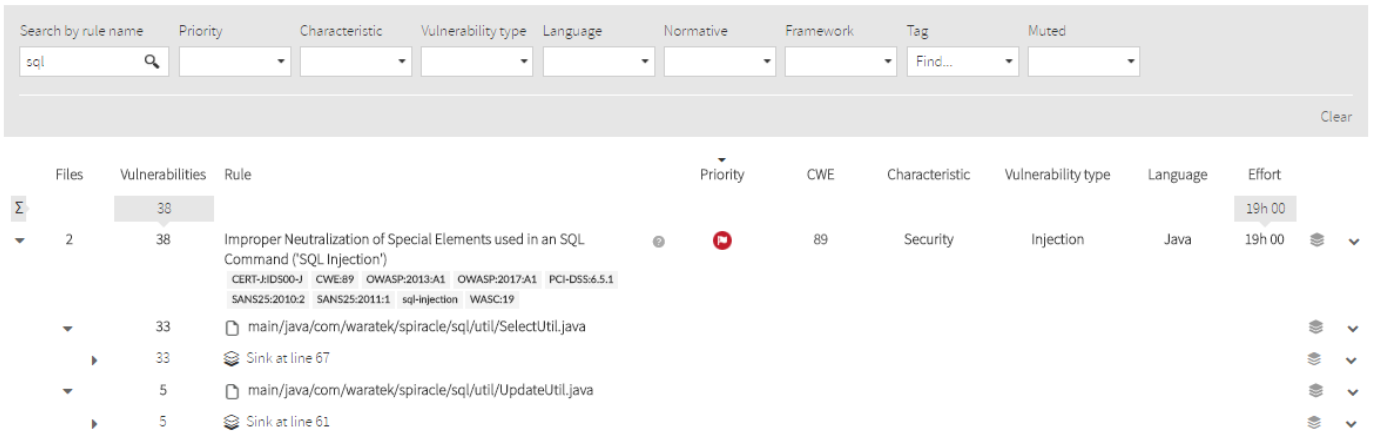
Below image shows SQl-Injection vulnerabilities. Just use the "Search by rule name" filter..



We can see in the image that there are 2 files affected, i.e. two files where there's a *sink* that is being feed by tainted data, i.e. an *injection point*.

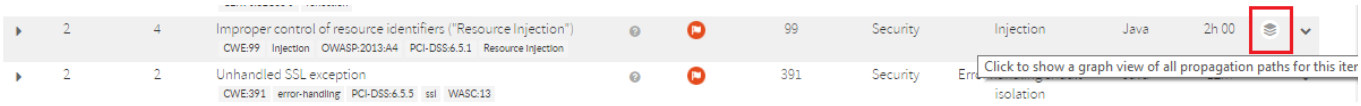
Also, we can see that for every file there are a number of vulnerabilities.

What do those vulnerabilities mean? See next image.

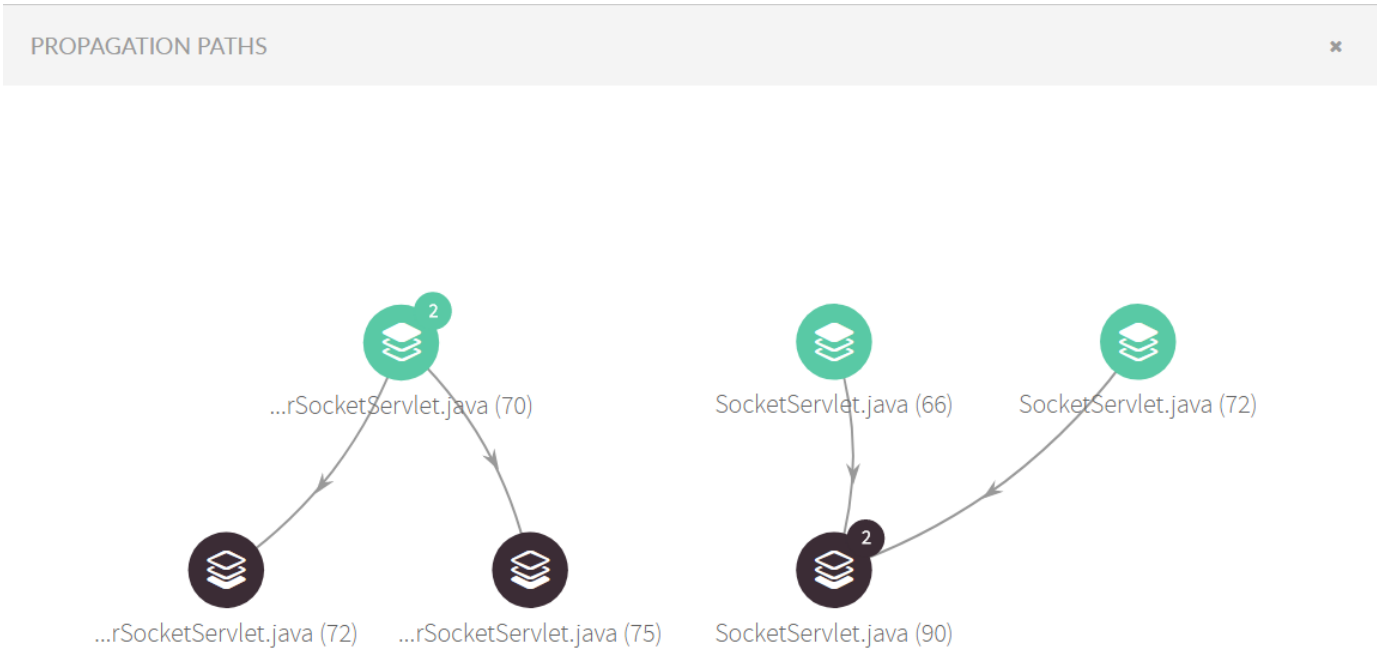


Graphical view of all Sources and Sinks related with a Vulnerability

For every vulnerability, you could see next icon



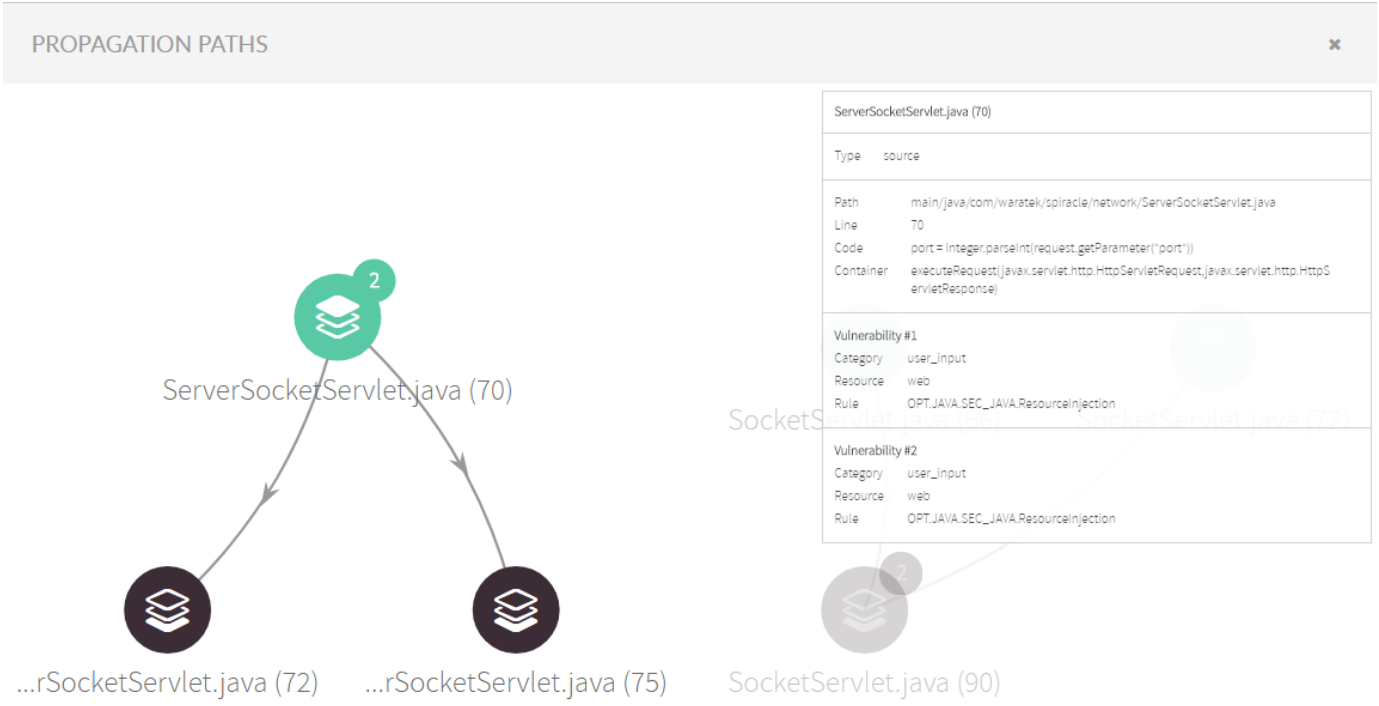
Clicking on that icon will open a graphical view of the complete set of sources, sinks and tainted flows of the selected vulnerability (rule) of your application.



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 which many tainted data flows participates.

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

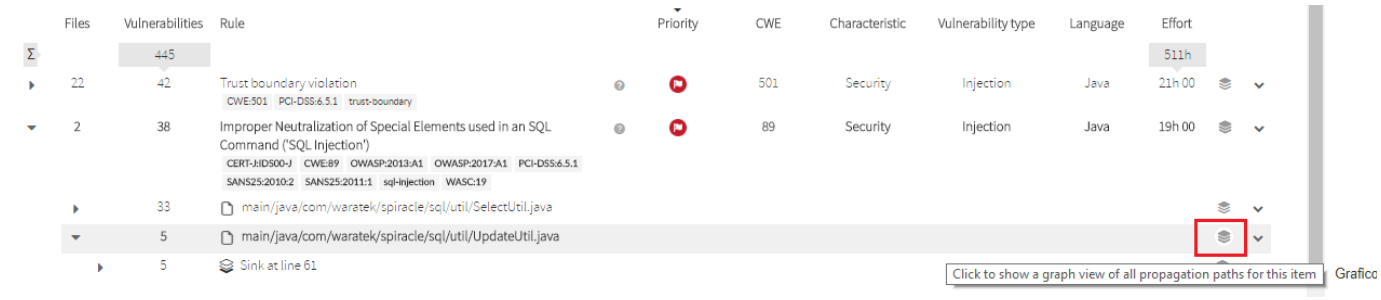


Finding Sources and Sinks

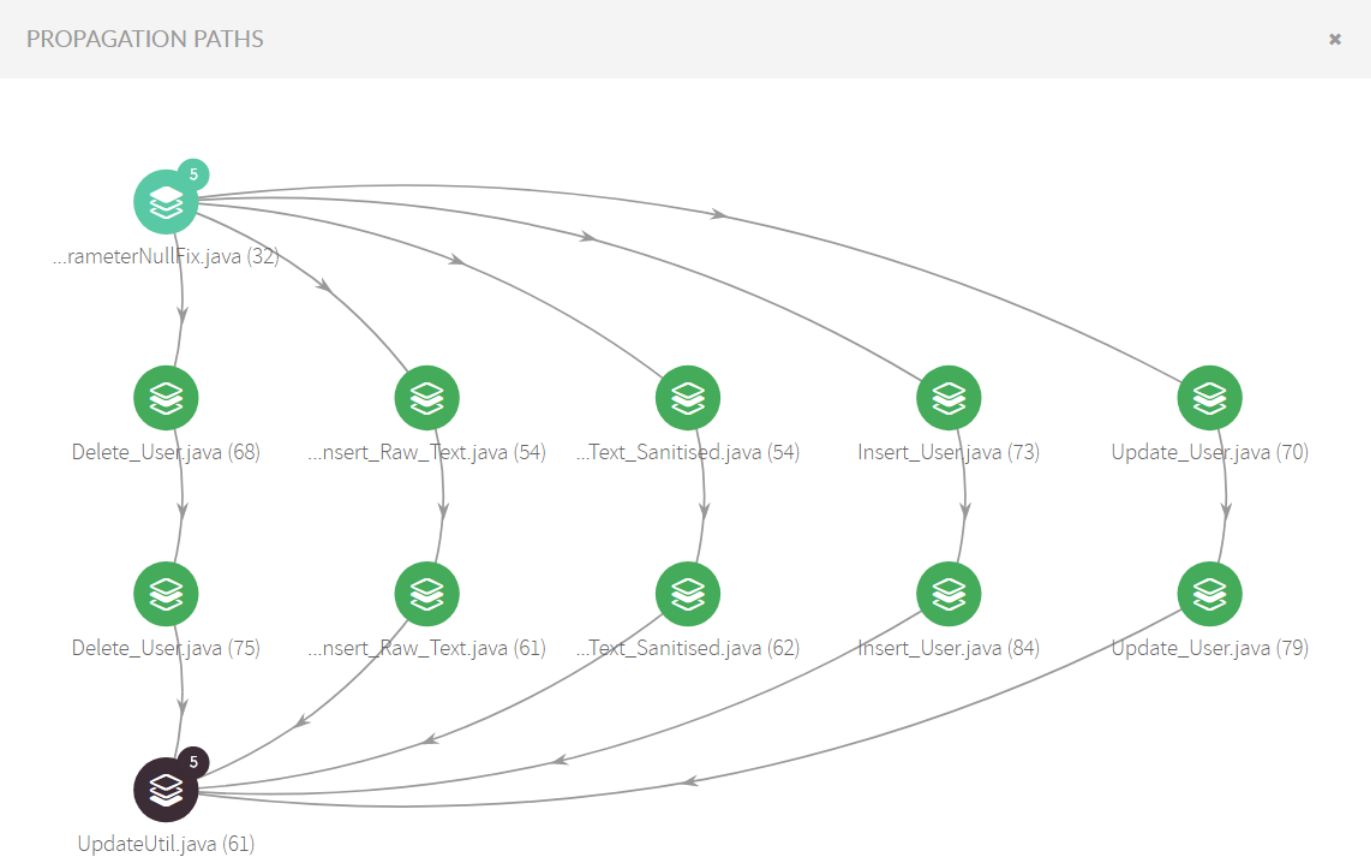
Clicking on the affected file will open all its affected sinks.

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.

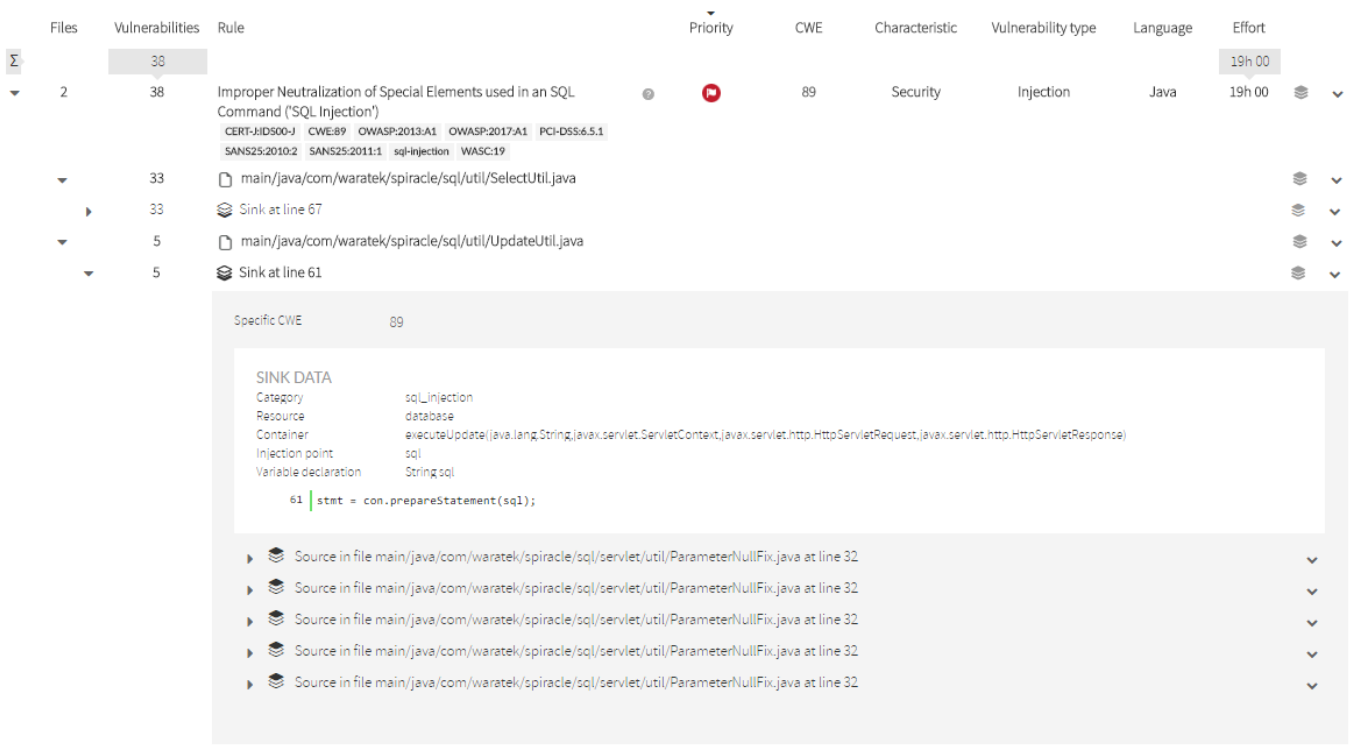
If you want to see a **grahical view** of all the sources, sinks and tainted data flows for **a file** you can also click on the icon.



And the following graph will be displayed.



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



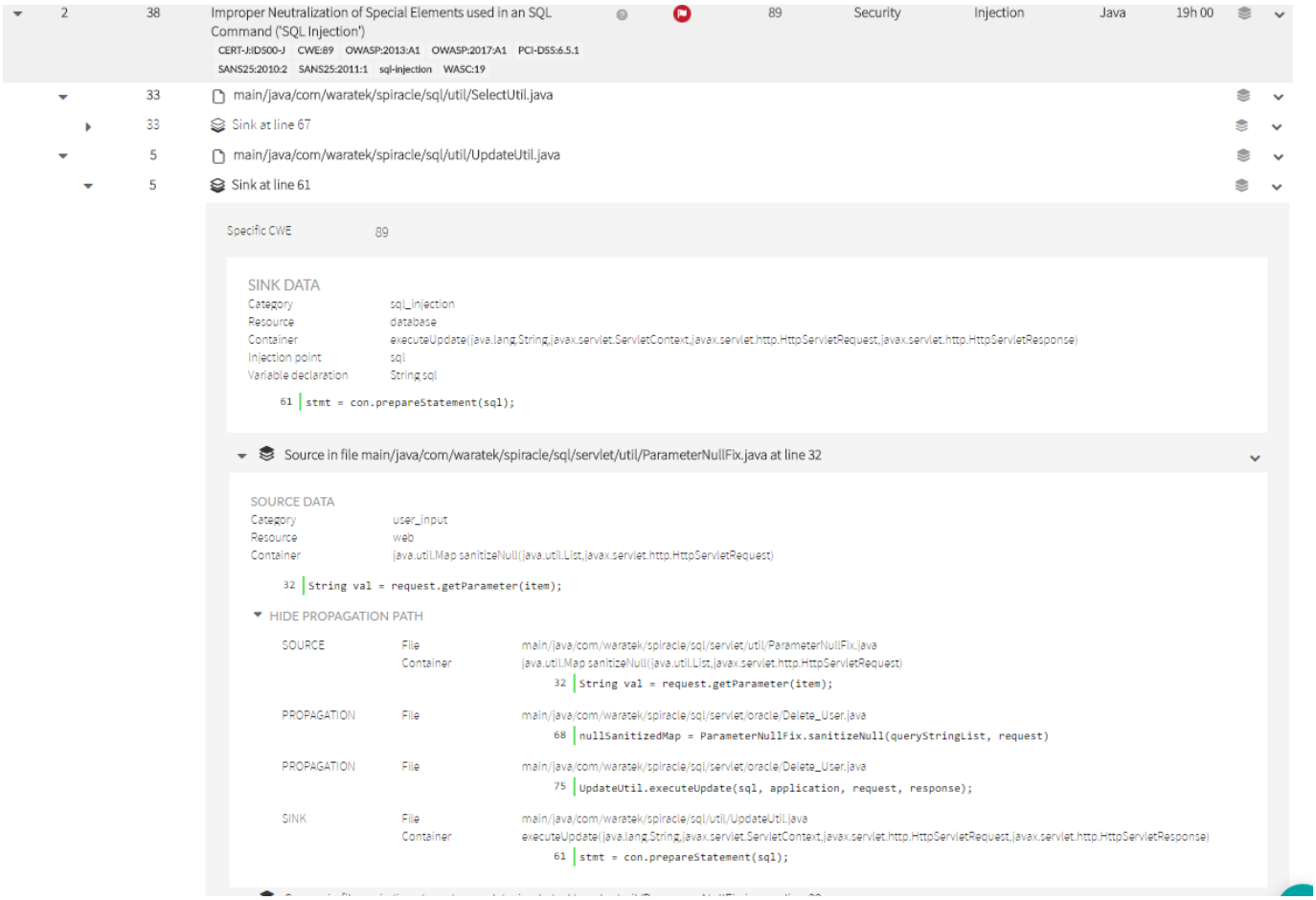
Sink's detailed information

Sink data
Sink detail includes 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 information about the source.

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 see with the example.

Sink at line 61

Specific CWE 89

SINK DATA

Category	sql_injection
Resource	database
Container	executeUpdate(java.lang.String,javax.servlet.ServletContext,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Injection point	sql
Variable declaration	String sql

```
61 | stmt = con.prepareStatement(sql);
```

Source in file main/java/com/waratek/spiracle/sql/servlet/util/ParameterNullFix.java at line 32

SOURCE DATA

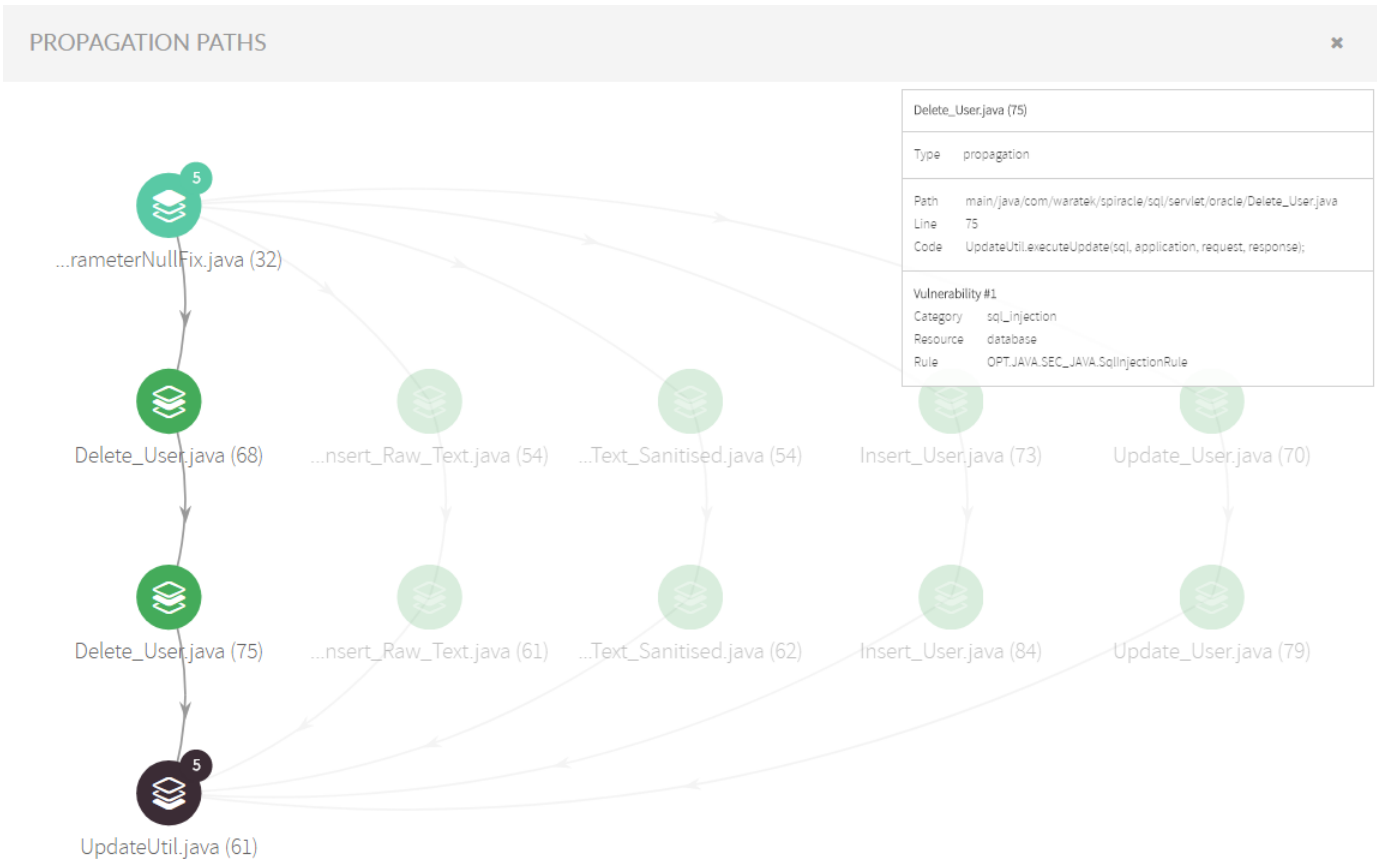
Category	user_input
Resource	web
Container	java.util.Map sanitizeNull(java.util.List,javax.servlet.http.HttpServletRequest)

```
32 | String val = request.getParameter(item);
```

HIDE PROPAGATION PATH

SOURCE	File	main/java/com/waratek/spiracle/sql/servlet/util/ParameterNullFix.java
	Container	java.util.Map sanitizeNull(java.util.List,javax.servlet.http.HttpServletRequest)
		`32 | String val = request.getParameter(item);`
PROPAGATION	File	main/java/com/waratek/spiracle/sql/servlet/oracle/Delete_User.java
		`68 | nullSanitizedMap = ParameterNullFix.sanitizeNull(queryStringList, request)`
PROPAGATION	File	main/java/com/waratek/spiracle/sql/servlet/oracle/Delete_User.java
		`75 | UpdateUtil.executeUpdate(sql, application, request, response);`
SINK	File	main/java/com/waratek/spiracle/sql/util/UpdateUtil.java
	Container	executeUpdate(java.lang.String,javax.servlet.ServletContext,javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpSe
		`61 | stmt = con.prepareStatement(sql);`

You can also view it in grahical mode:

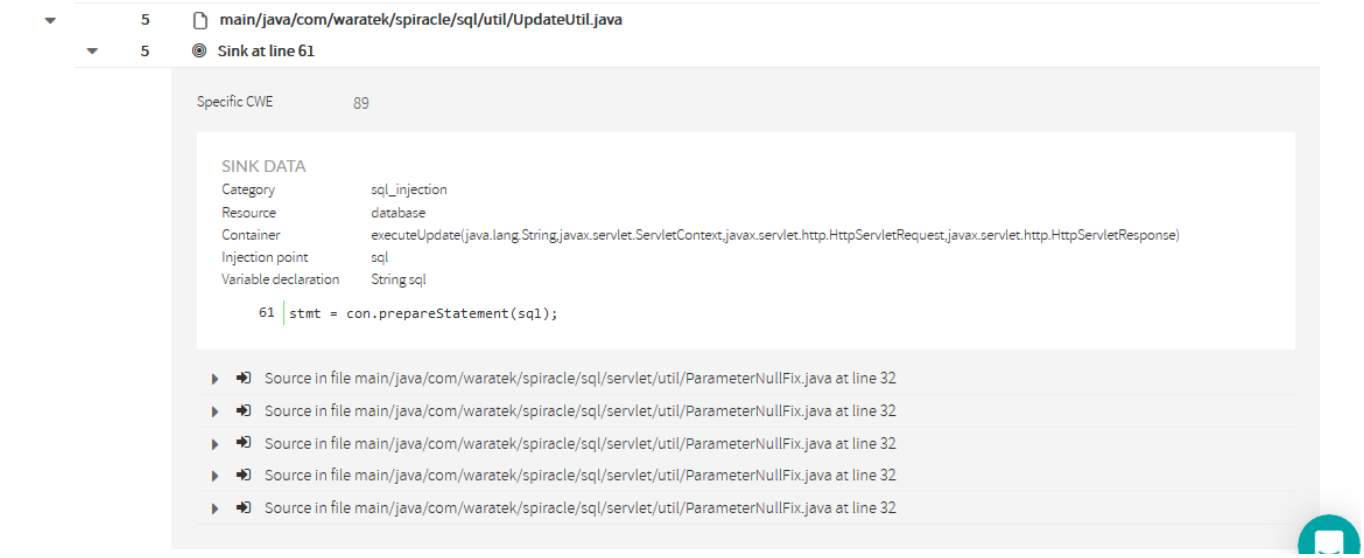


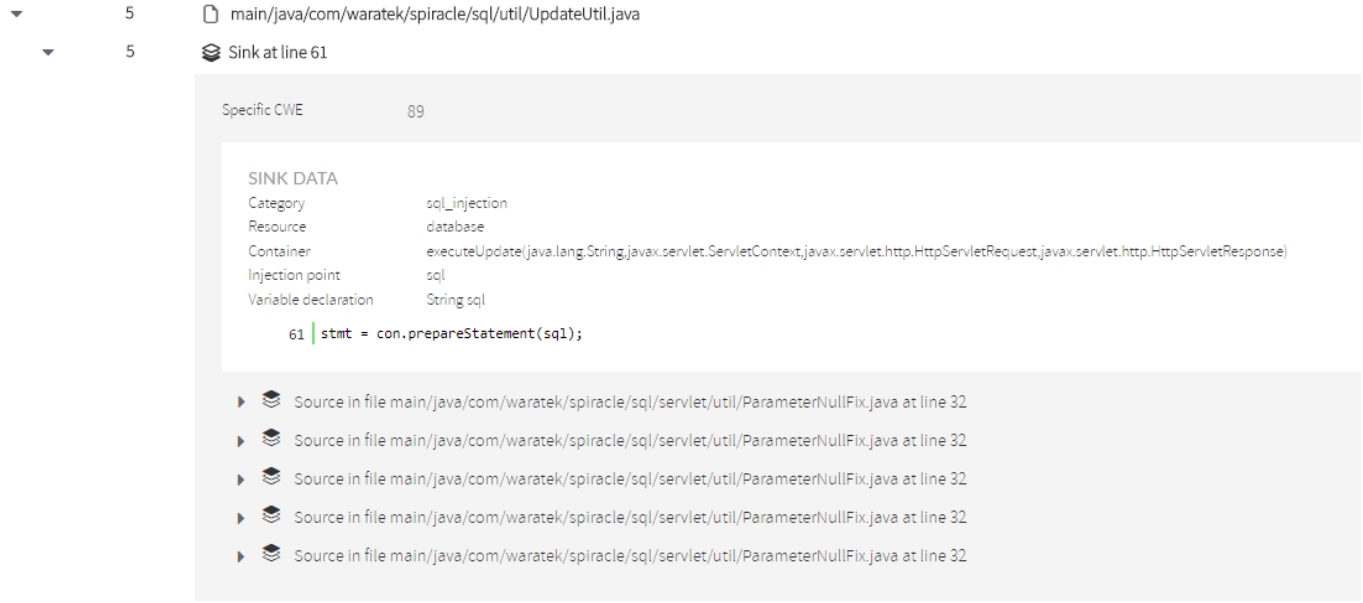
Any **propagation path** is composed of a **source node**, a **sink node** and as many **propagation nodes** as different methods are involved in the propagation path.

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

- Source node indicates that 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.
- First propagation node indicates that *Delete_User.java* line 68 receives back the above tainted data.
- 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.
- Sink node indicates that file *UpdateUtil.java* line 61 (within *executeUpdate(…)* method) injects tainted data (sql sentence) to a PreparedStatement that finally is executed against the database.

Once explained a concrete source-sink propagation path, let's go back to the initial sink information.

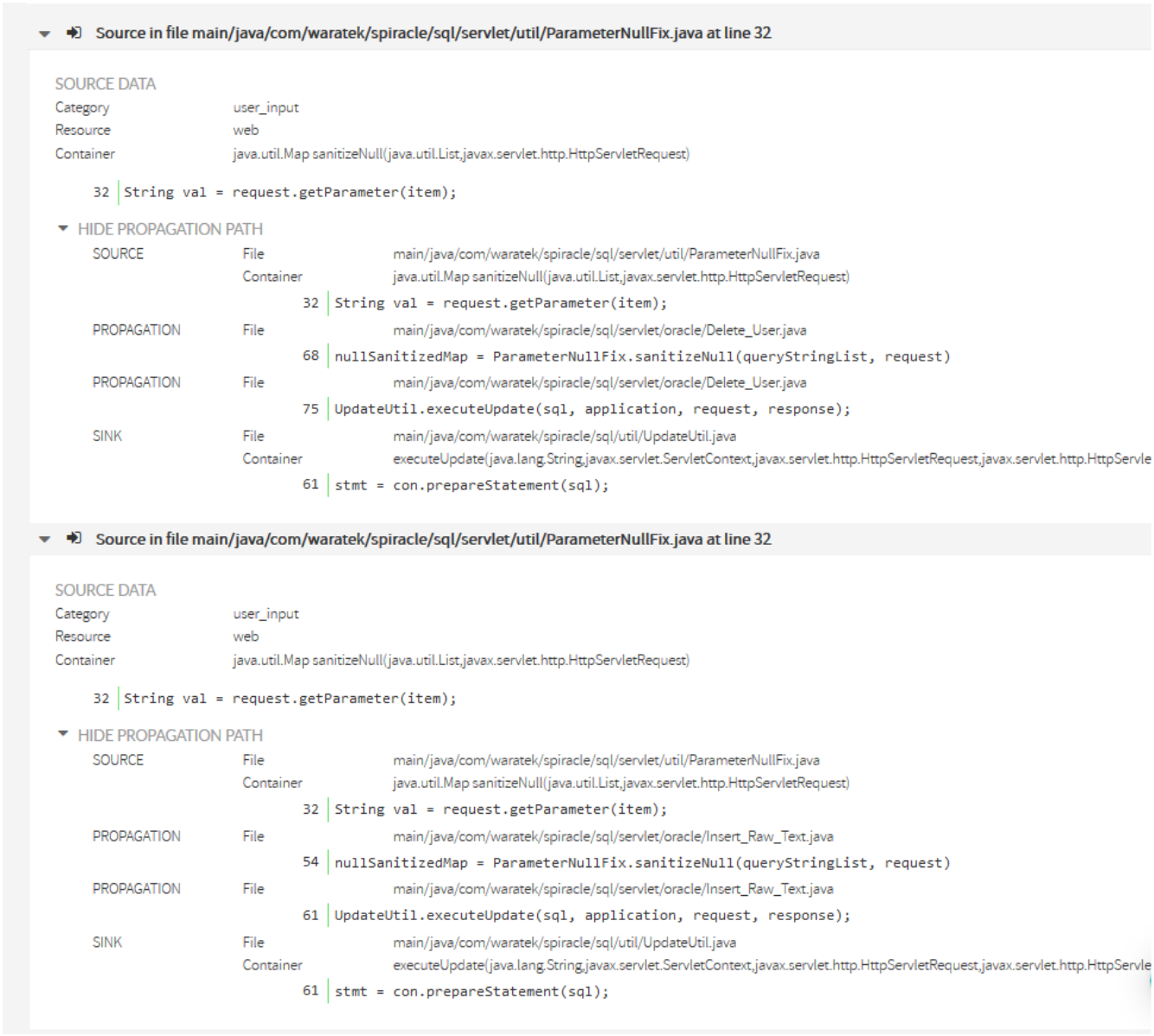




As said above, for every sink there will appear the list of sources that are “feeding” that sink.

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

We’ve selected this specific example because it shows something that would be possible to happen in your own code. Let’s go in detail exploring the 2nd source.



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

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

Remainder sources for the same sink shows that there are still other different propagation paths between same source and 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 a SQL sentence with those 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 source and sink be the same.

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. 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 to 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.