RuleKeeper 2022 Usability Tests

1 Introduction

First of all, thanks for participating! The goal of this study is for me to understand what are the main difficulties in using my tool - RuleKeeper. So, first, I'll give you a brief overview of the tool and provide you with an example for you to understand the pipeline. Next, I'll ask you to perform two simple tasks, similar to the ones in the example, but with a different use case.

Setup

Before starting, we need to make sure you'll be able to run RuleKeeper. To be able to run the experiments, I provided you with a Vagrantfile. To set up the machine, you'll need to have vagrant and VirtualBox (provider) installed. To check if you have vagrant installed run: `vagrant -v`. To check the available providers, run: `vagrant plugin list`.

Now, assuming you have vagrant installed, let's start by setting up the machine. To do so, run: `vagrant up` in the directory that contains the `Vagrantfile`. This may take a while. As a sanity check, please show me the output of this command.

To access the machine, you can run the command: `vagrant ssh`. This will only be needed later in the guide.

2 What is RuleKeeper?

In simple terms, RuleKeeper is a GDPR-aware policy enforcement system for web frameworks that can help web developers to catch application bugs that would result in a violation of their GDPR privacy policy. The privacy policy is the document that specifies how personal data is going to be processed by the application. RuleKeeper operates in two phases: the *offline* phase and the *runtime* phase. In this test, we'll focus only on the *offline* phase.

The *offline* phase takes place at development time before deploying the web application to production. In this phase, the web developer specifies a *GDPR manifest* to describe how he expects the personal data of users to be processed by his web application. Simply put, this *GDPR manifest* is an adaptation of the organization's privacy policy, written in English, that specifies the personal data types and purposes for which the web application can process the data. An example of this manifest is provided ahead. RuleKeeper implements a static code analysis pipeline that allows the web developer to verify if the GDPR manifest he wrote reflects the way that the application is behaving.

In the next section, I'll provide you with an example of a toy application to explain each of the steps of this preliminary analysis.

3 The Webus example

Webus is a hypothetical toy web application that offers a bus booking service, maintained by a team of developers. Webus allows users to browse bus schedules, buy tickets and subscribe to a newsletter.

3.1 The web application

In this work, our emphasis is given to classical 3-tier JavaScript web-based architectures, composed by presentation, logic and data layers, as described next.

Presentation layer: Consists of web pages and JavaScript code running on the browser, and it is implemented with React.js.

Logic layer: Consists of JavaScript server-side code. This code exposes an API for serving HTTP requests to Webus operations by the presentation layer. The API is composed of a route per operation (i.e., a specific URL) and each operation is implemented by a piece of JavaScript code, named controller. Figure 1 illustrates the code for the *subscribe to newsletter* operation. As we can see, it is composed of a route (`/subscribe`) and a piece of code that inserts the new e-mail into the database.

Data layer: Uses MongoDB, a non-relational DBMS, for storing persistent data in databases. Each database is laid out via a schema. In Webus, the data is organized in three tables, containing the information suggested by their respective names: `Schedules`, `Tickets`, and `Newsletters`.

```
router.post('/subscribe', (req, res) => {
    const { e_mail } = req.body;
    if ( !e_mail) { res.status(400).json('Missing information'); return; }

    /* Create subscription e-mail object */
    const subscription = { e_mail: e_mail };

    /* Add new e-mail to newsletter database */
    newsletters.create(subscription,  (err, subscribedEmail ) => {
      if (err) { res.status(400).json('Error subscribing.'); return; }
      else res.status(200).json({ subscribedEmail });
    });
})
```

Figure 1: Code snippet of the *subscribe to newsletter* operation.

3.1.1 Webus code

Webus code is located in the `webus/` folder. If you are having trouble accessing the code in the virtual machine, please inform me and I'll provide it to you. All of the business-related code are located in a single file – `app.js` – that contains all routes for the Webus operations, as described below:

Operation	Endpoint
Subscribe to newsletter	`POST /subscribe`
See schedules	`GET /schedules`
Buy ticket	`POST /buy_ticket`

Table 1: Webus operations mapped to endpoints

The code related to the database schema is located in the `webus/models` folder. Here, we find three files, corresponding to the schemas of the three database tables: *Schedules*, *Tickets*, and *Newsletters*, as described below:

File	Database	Columns
`schedules.js`	Schedules	`destination, date`
`tickets.js`	Tickets	`name, destination, date, credit-card`
`newsletter.js`	Newsletter	`e_mail`

Table 2: Webus data tables.

Please, take your time to analyze the code and make sure you understand how it operates. Any doubt here, please tell me.

3.2 Privacy policy

To abide by the GDPR, the Webus provider must elaborate a privacy policy, written in English, to specify the personal data and purposes for which the web application can process the collected

data. In this case, Webus collects several pieces of personal data of its users and different data is used for two main purposes: *ticket management* purposes, being materialized by the operation *buy a ticket* and *marketing* purposes, which are put into action by operation *subscribe to newsletter*. A simple English privacy policy for Webus could be:

> The Webus application collects the personal data of its customers, namely, the user's name, e-mail, and credit card. Other data that is considered personal, since it can be used to identify the user is the ticket's destination and date. We process the user's name and credit card and the ticket's destination and date when the user is buying a ticket, for **ticket management** purposes. We process the user's email when the user subscribes to the newsletter, for **marketing purposes**. The lawfulness base for both purposes is the user's consent.

3.3 GDPR Manifest using RuleKeeper

To specify the privacy policy in a non-ambiguous and expressive way, developers write a GDPR manifest, using our language. Here, I'll explain this idea using an example of a GDPR manifest that a developer would write for Webus.

3.3.1 Application plane

The developer starts by specifying the GDPR-agnostic properties of the app. To describe this plane, the developer will first look at the application code, in our case, described in Section 3.1.

1. Data Items
Defines labels of all data types that can be collected and processed by the web application. In our example, as we can see from Table 2, Webus processes seven data types: the user's name, credit card, and email, the tickets' date and destination, and the schedules' date and destination. So, we define `data-item()` rules:

```
data-item(user name)
data-item(user credit card)
data-item(user email)
data-item(ticket destination)
data-item(ticket date)
data-item(schedule destination)
data-item(schedule date)
```

2. Data Mapping
This field maps the data items to the corresponding database schema, particularly the table and column where it is stored. So, we define `map-data()` rules as `map-data(<data>, <table name>, <column name>)`:

```
map-data(user name, tickets, name)
map-data(user credit card, tickets, credit_card)
map-data(user email, newsletter, e_mail)
map-data(ticket destination, tickets, destination)
map-data(ticket date, tickets, date)
map-data(schedule destination, schedules, destination)
map-data(schedule date, schedules, date)
```

3. Operations
Data, whether personal or not, is processed by the application through specific operations. The `operation()` rule defines labels of the existing operations. Webus has three, as described in Table 1.

```
operation(see schedules)
operation(buy ticket)
operation(subscribe to newsletter)
```

4. Operation Mapping
Operations are mapped to the corresponding web endpoints exposed by the web application, using the `map-operation()` rule, as `map-operation(<operation name>, <endpoint>)` as defined below:

```
map-operation(see schedules, GET /schedules)
map-operation(buy ticket, POST /buy_ticket)
map-operation(subscribe to newsletter, POST /subscribe)
```

3.3.2 GDPR plane

The next step is to specify the attributes related to the GDPR. To describe this plane, the developer will look at the privacy policy, written in English – in our case, described in section 3.2.

1. Personal Data
Data types can be classified as personal data. As observed in the privacy policy, Webus defines as personal the data user's name, credit card, and email and the tickets' date and destination as personal. So, we define `personal-data-item()` rules:

```
personal-data-item(user name)
personal-data-item(user credit card)
personal-data-item(user email)
personal-data-item(ticket destination)
personal-data-item(ticket date)
```

2. Purposes
The policy defines the purposes for which data can be processed. Webus collects data for two distinct purposes: ticket management and marketing. So, we define `purpose()` rules:

```
purpose(ticket management)
purpose(marketing)
```

3. Data Collection
The data that can be collected and further processed for each purpose is also described in the privacy policy. Each purpose cannot process data that was not initially collected for it. So, we define `collected-for()` rules as `collected-for(<purpose>, [<data items>])`:

```
collected-for(ticket management, [user name, user credit card, ticket destination,
↪  ticket date])
collected-for(marketing, [user email])
```

4. Lawfulness Base
Additionally, each purpose must be associated with a reason for its processing. In the context of our work, lawfulness bases are the users' consent. So, we define `lawfulness-base()` rules as `lawfulness-base(<purpose>, "consent")`:

```
lawfulness-base(ticket management, consent)
lawfulness-base(marketing, consent)
```

5. Operation's purposes
Each operation is executed for a purpose. As written in the privacy policy, Webus *subscribe to newsletter* operation is executed for marketing purposes and *buy ticket* is executed for ticket management purposes. So, we define `executed-for` rules as `executed-for(<operation name>, <purpose>)`:

```
executed-for(buy ticket, ticket management)
executed-for(subscribe to newsletter, marketing)
```

The GDPR manifest is the combination of all these rules.

3.4 Running RuleKeeper

As I said before, the goal of RuleKeeper is to verify if the GDPR manifest reflects the way that the web application is processing data. To do this, RuleKeeper will first build two files: the parsed manifest and the static analysis output. To run RuleKeeper with Webus, you'll need to set up the environment. This outputs the static analysis output and is only done once per use case. To do that, run the following commands, in the directory of the `Vagrantfile`:

```
vagrant ssh
cd tests
./setup.sh webus
```

This command may take a while. It runs in the background, so when it reaches the below output, please press enter to continue.

```
INFO  Bolt enabled on 0.0.0.0:7687.
INFO  Remote interface available at http://localhost:7474/
INFO  Started.
```

Next, to generate the parsed manifest file, you need to run RuleKeeper, as described below. This can be done every time you want to check if the manifest you wrote is correct (no need to run the setup again).

```
./run.sh webus
```

This command may take a while. The expected output is:

```
> rulekeeper-parser@0.0.1 start
> ts-node index.ts

Difference:
 Set(0) {}
```

If the output is not this, please let me know.

The *parsed manifest* processes the rules provided in the manifest, and associates each operation with the **allowed** data. This file is saved as `gdpr-manifest-parsed.json` in the use case folder. In the Webus case, is saved as `webus/gdpr-manifest-parsed.json`. Please, take your time to analyze the file and make sure you understand what it represents. Any doubt here, please tell me.

The *static analysis output* is the result of processing the application code. It uses the same representation as the *parsed manifest*, associating each operation with the data processed by it (by the queries). This file is saved as `app-analysis-result.json` in the use case folder. In the Webus case, is saved as `webus/app-analysis-result.json`. Please, take your time to analyze the file and make sure you understand what it represents. Any doubt here, please tell me. With

the exception of the `GET /schedules` endpoint, the files match. This happens because the static analysis detects the operation *get schedules* but since it does not process personal data, it is not mentioned in the policy. These cases are ignored by our tool since they are not relevant for the analysis. As you can see, all the other endpoints match, meaning that the application is processing the data according to the manifest. This match can be checked by the previous output of the tool: `Difference: Set(0)` .

4 Usability Tests

4.1 Case study: LAB

LAB is an application that offers a clinical analysis service, maintained by a team of developers. Next, I'll describe both the application code and the privacy policy, as I did with Webus.

Web application

LAB code is located in the `lab/` folder. All of the business-related code is located in a single file – `app.js` – that contains all routes for the LAB operations, as described below:

Operation	Endpoint
Register user	`POST /register_user`
Enroll patient	`POST /enroll_patient`
Consult patient data	`GET /:patientId`
Update patient data	`POST /:patientId`
Subscribe to newsletter	`POST /subscribe`

Table 3: LAB operations mapped to endpoints

The code related to the database schema is located in the `lab/models` folder. Here, we find three files, corresponding to the schemas of the three database tables: *Users*, *Patients* and *Newsletters*, as described below. Note that there are two different `e_mail` instances: one associated with the users' registration (stored in the *Users* table) and the other associated with the newsletter (stored in the *Newsletters* table).

File	Database	Columns
`users.js`	Users	`username, password, e_mail, registration_date`
`patients.js`	Patients	`citizen_card, patient id, full_name, birth_date, gender, ss_number, photo, address, mobile_number`
`newsletter.js`	Newsletters	`e_mail`

Table 4: LAB data tables.

Please, take your time to analyze the code and make sure you understand how it operates. Any doubt here, please tell me.

Privacy policy

To abide by the GDPR, the LAB provider elaborated a privacy policy – written in English, as described below.

The LAB application collects personal data of its patients, namely, the user's username, password, email, registration date, citizen card, patient id, full name, birth date, gender, ss number, photo, address, mobile number, and newsletter email. We process the user's username, password, email, and registration date when the user is registered, for user management purposes. We process the user's citizen card, patient id, full name, birth date, gender, ss number, photo, address, and mobile number when the patient is enrolled, when the patient is updated, and when the patient data is consulted, for clinical analysis purposes. We collect the user's newsletter email when the user subscribes to the newsletter, for marketing purposes. The lawfulness base for all purposes is the user's consent.

Task 1: Write the GDPR manifest

The first task is to write the GDPR manifest of LAB, considering the provided application code and privacy policy. We encourage you to look at Webus and follow the example. Use the file `lab/gdpr_manifest.txt`, already pre-filled with the rules. When you reach this point, tell me, so that I can start measuring the time.

To setup RuleKeeper, run (once):

```
./setup.sh lab
```

To check if the manifest is correct, run RuleKeeper (every time you want to check):

```
./run.sh lab
```

If the manifest matches the application code, the output is:

```
> rulekeeper-parser@0.0.1 start
> ts-node index.ts

Difference:
 Set(0) {}
```

If the `Set` is not empty, it means that at least one operation is different – i.e, the manifest says that the operation can only process some data, but the application code is processing more. Please, check both generated files: `lab/gdpr-manifest-parsed.json` and `lab/app-analysis-result.json` to compare the differences. When you finish, please tell me, so I can measure the time, and please save the `lab/gdpr_manifest.txt` to send me at the end of the experiments.

Task 2: Find the bug

Suppose LAB developers implemented a new marketing feature to send promotional codes to the oldest users (registered prior to 2020) if they subscribe to the newsletter. So, besides the newsletter e-mail, the subscription logic will also need to process the user's information as well, namely, the registration date. The new version of the application is located in `lab_v2/` folder. All code is the same, except the *subscribe to newsletter* operation.

1. Run RuleKeeper again, with the new version, as shown below. Can you detect the changes between the files `lab_v2/gdpr-manifest-parsed.json` and `lab_v2/app-analysis-result.json`?

```
./setup.sh lab_v2
./run.sh lab_v2
```

2. After running the previous commands, a copy of your `lab/gdpr_manifest.txt` will be copied into the `lab_v2/` directory. Fix the `gdpr_manifest.txt`, in order to support this new change. You

can change any of the rules, except removing personal-data-items. The output of the tool must show an empty set: `Difference:  Set(0)`

When you finish, please tell me, so I can measure the time, and please save the new `lab_v2/gdpr_manifest.txt` to send me at the end of the experiments.