SeQG - A Security Question Generator as a Webapplication

Project Documentation
of

@Finn Kock, @Alejandro Castilla,
@Sergio Pajuelo , @Robert Eggert

Ludwig-Maximilians-Universität München
Practical Course Usable Security

Reviewer:	Dr. Viktorija Paneva and Doruntina Murtezaj
Professor:	Prof. Florian Alt

30.07.2025

1 Index

...

2 Introduction

This is the intro to our project

3 Frontend

this will get split up later

4 Backend

The Backend is the silent core of the whole project, which makes it possible to create the connection between the user and front-end. As the user establishes a connection to the front-end the next Backend component is in use, namely the interaction between the front-end and the LLM API After a session our backend then also covers the safe disconnection logic and resets everything to a clean state.

4.1 QR Code - Connection

The whole connections starts by making sure the user visits the link via. scanning the QR code. Afterwards a link gets generated, that is unique for every session and contains a JWT token. This token gets generated by the backend and send over to the front-end and only then the user may connect. In that way, we make sure that no connection can be established without the user having the right to do so. The front-end then uses this exact JWT token and only allows connections with the same token in current use. Considering all previous steps, the actual connection logic is then handled by the users front-end, from which the device (e.g. phone) emits a ”register” message to the backend. This leads to the final step of the connection, where the backend receives the message and sends over the status that is appropriate for the current connection state. If everything worked optimally the front-end gets the message ”connected” emited from the backend.

4.1.1 Role & Status

First we have to define what actual roles can connect, this makes random or brute connections even harder. Currently we have two roles, namely the ”client” and ”host” role. The client is the one who connects to the host, whereas the host is the one who gives the opportunity connecting to the currently active session.

To make sure a connection is actually valid we now discuss the part where we need to define different states helping us understanding what went pottentially wrong in a connection.
The following states currently in use are defined as:

- **pending** - The QR Code is created and the front-end is waiting for the user to connect.
- **connected** - The user has successfully connected to the session and can now use the application as intended.
- **disconnected** - The user has disconnected from the session, either by closing the application or by a timeout.
- **already_connected** - The user or a host was already connected to the backend, which moves potential intruders into a url sinkhole.

The states that should be added for a more detailed error handling are:

- **no_host** - The user tries to connect to a session that does not exist.
- **invalid_role** - The user tries to connect with a role that does not exist.

4.1.2 Registration

Making the connection valid though, it is not enough to just scan the qr code. We actually need a registration logic that makes sure that the host and client are allowed to connect. This is done by the host front-end and clients front-end, who both need to emit a ”register” message to the backend. By registering, the backend recognizes if a host or client is already connected. This makes flag handling easier, as the backend can then emit a message to the host or client front-end appropriate to the actual status. If the user is in private mode, then we emit a special message called ”register-private”, which then creates a file with a unique uuid and a token in the local storage of the users browser. This file and local storage token is only created though if the user submits his age and experience first. Such a file would then look like this:

```
{
    "user_id": "abc123",
    "age": "< 18",
    "experience": 2,
    "progress": {}
}
```

Where the **user_id** (String) is the unique identifier for the user, **age** (String) is the apporximate age of the user, **experience** (Integer) is the experience level of the user and **progress** (Object) is an object that will hold the progress of the user with counted questionTypes and if they where answered correctly.

4.2 API Calls

In actuality currently there exist four API calls the qr-code backend is handling.

- **/connect/host** - This is the call that is used by the host to connect to the backend and receiving the JWT token.
- **/disconnect/all** - This call is used by the client or host to close the current session.
- **/private/user-data/:userId** - Here we get the private-user data from the backend, which is already stored (currently only age and experience).
- **/private/saveAgeAndExprience** - This call is used to save the age and experience of the user in a private session.

Adding more API calls is possible, but currently not needed. If one shall adjust those calls then only in the matter of better verification, so that no workaround allows to use these API calls without the right to do so.

4.3 LLM - Connection

Before the user gets to actually answer questions provided by the LLM, we need to submit our age and experience, if we did not already do so. Only then the questions will start to get fetched from the LLM API. But this is not happening directly, for that we also implemented a little backend, that needs to be started, which then handles all needed API calls. Currently though the backend is not protected against calls from outside, so it is not recommended to run it on a public server. One might first want to implement some kind of authentication, before doing so. The same goes for the LLM API, as it is a Ollama Backend that runns openly. Here we also recommend to implement authentication procedures such as private keys or doing network segmentation only for the API to prevent unwanted access.

4.3.1 Tips

To catch a potential users attention, we decided to use Tips and eye catching messages on our main screen. This shall then make the users interest about our PSUI rise and let him or her approach our Public Display. These tips are getting fetched from the LLM API. Incase of server connection issues, we have a fallback mechanism that uses a locally saved tips. Later on in the future, one may also add not only tips but also current news in cybersecurity and generally speaking of security awareness.

4.3.2 Questions

Here we differentiate between the two modes as one of them has a userId and the other one does not. Considering that, lets first look at the guest-mode question api request. For that we have prepared a prompt and a list of topics from which the llm can choose. Both of them are stored in the backend folder. From here on we directly pass arguments into the prompt such as age, experience and the topic. When we are done, we simply send over the whole prompt to the LLM API, which then responds with a specific json format. The json format is then parsed and send over to the host front-end. Formatted the json represents this structure:

```
{
    "question": ... ,
    "option_s": ... ,
    "dropZones": ... ,
    "correctAnswer_s": ... ,
    "topic": ... ,
    "questionType": ...
}
```

Each one having there own meaning.

- **question**: The question that the user has to answer (String).
- **option_s**: The options that the user can choose from (String array).
- **dropZones**: Represent drop zones in a drag & drop event, where the user can drag and drop the options (String array).
- **correctAnswer_s**: Correct answers for the question (String array).
- **topic**: The overall topic of the question (String).
- **questionType**: The event-type of the question (String).

4.3.3 Explanation

In the explanation we actually dont need to differentiate between the two modes, as the explanation should be universal. There may be a critical moment where one might think to add some extra explanation features for the private mode but we did not implement that. As we have fetched a question from the LLM API we can refeed it with the almost same provided json format. In our case we only need the question and the correctAnswer_s aswell as a boolean telling the LLM we did indeed answer the question incorrectly.
When we do answer correctly there is no need to explain the question as the user already knew the answer (hopefully did not guess). This means we have a simple return format for our explanation, because in the end we only need the explanation text.
Therefore our json format looks like this:

```
{
    "explanation": ...
}
```

Where the **explanation** (String) is the text that the LLM provides us with.

4.3.4 Saving Answers

When the user enters the guest mode, we do not save anything but rather do a local session with grading at the end. However, when the user enters the private mode, we do save the answers so that his or her previously answered questions can be taken into account. This makes the LLM way more personalized and the user can also see his or her progress eventually. The main idea is, that we get the questionType from the LLM API. After that we add to our autogenerated file with the userId as its filename the current questionType. For the final part, we increase a counter for the correctly answered questions according to the questionType. If the user indeed answered the question correctly, we increment by one, otherwise not. The total amount of questions is also

saved for each questionType. This amount always gets incremented by one, no matter if the user answered correctly or not.
In general a progress output may look like this:

```
{
    ... ,
    "progress": {
        "passwords": {
            "correct": 0,
            "total": 1
        },
        "camera_microphone_access": {
            "correct": 0,
            "total": 1
        },
        "kids_online_safety": {
            "correct": 2,
            "total": 2
        }
    }
}
```

Which would mean that in total the user has answered 4 questions, of which he answered 2 correctly.

4.3.5 Feedback

The moment the user decides that he or she has had enough of the questions, one can request a feedback. This feedback is then generated by the LLM API and sent over to the front-end. The user has now the oppertunity to read the feedback and then decide for himself learning more about a topic or not. Feedbackwise we offer a spider chart limited to the maximum amount of 8 topics, where we define from 0 to a 100 percent how well the user did in each topic. Aswell as a specificly generated text, that gives the user a more detailed overview of his or her performance. Afterwards the user can disconnect from the session or do nothing as the session will automatically disconnect after a certain amount of time.

4.4 Anti-Breaching Sessions

As every server, that is reachable from the internet or inside a local network, we need to make sure, that we are as secure as possible. For that we implemented a simple anti-breaching session logic. Everytime a user connects to the backend we check if someone is already connected (max amount of client: 1, host: 1). Therefore moving the unwanted user into a url sinkhole. This means that refreshing the page will keep you on a prohibited url where one cannot do anything, which eases the backend servers capacity. Additionally to that we have a structure, where even if an attacker gets the information about the current session, he cannot make anything of it. The actual data that is stored from the user does not contain any sensitive information, but rather a unique userId, a rough approximate of the age aswell as the experience level, the user provided us with. Additionally the JWT token is only valid for a certain amount of time, which makes it even harder to breach the session. Though our application is not perfect and we do not claim it to be, we still try to make it as secure as possible. One can for example try to impersonate a user by storing the userId in his or her own local storage, this would mean, that the attacker may use the interface as the pronounced user. However the attacker also has to be motivated to then answer questions for the user, which he or her is impersonating. This has no actual benefit for the attacker. We also do something that is almost bullet proof, so that even if an attacker mages to connect to the session there is nothing the attacker can do. As the QR Code is generated the user scans it. In that Moment the file is being created or reused from the connected user. Even if the attacker tries to connect now he or her will be send to the sinkhole as we remember the first user that connected with his or her specific userId. Therefore making sure the file of the user is safe and not touchable by a potential attacker.

4.5 File Cleansing

As the user logs into the private mode, we create a file, but how do we remove it again? There are alot of ways one could do so, but most of them require some interaction. Therfore we decided that a simple python script, running maybe once a month, shall remove all files that are older than 30 days. This may seem unfortunate for the user as his progress is lost after one month but for the servers memory sake we need a way to remove files after a certain amount of time. If we try to keep them forever this may lead to dead files, as users may never return or users often delete there browser cache, maybe even change there browser. All that has influence on how the files are getting accessed and therefore we really need a way to remove them. The script itself is in the backend folder called ”checkFiles.py”.

4.6 Contributors

@Finn Kock: ...
@Alejandro Castilla: ...
@Sergio Pajuelo: ...
@Robert Eggert:

- The Whole backend logic, including the LLM API, QR Code and the Anti-Breaching Sessions idea (except the Feedback api call).
- File based Routing and the File Cleansing logic.
- Most of the State and session management and the Connection/Disconnection logic.
- All question Types + overlay
- Touch compatability
- Configs