2025

Programming exercise report

COMP.SEC.300-2024-2025-1 SECURE PROGRAMMING

RIKU RUUSULAAKSO

Table of contents

1. Description of the program

The application is a combat tracker, meant for tabletop roleplaying games. It has the ability to save and load participants of a combat, be they players or nonplayable characters controlled by the gamemaster, such as enemies and friends the players have.

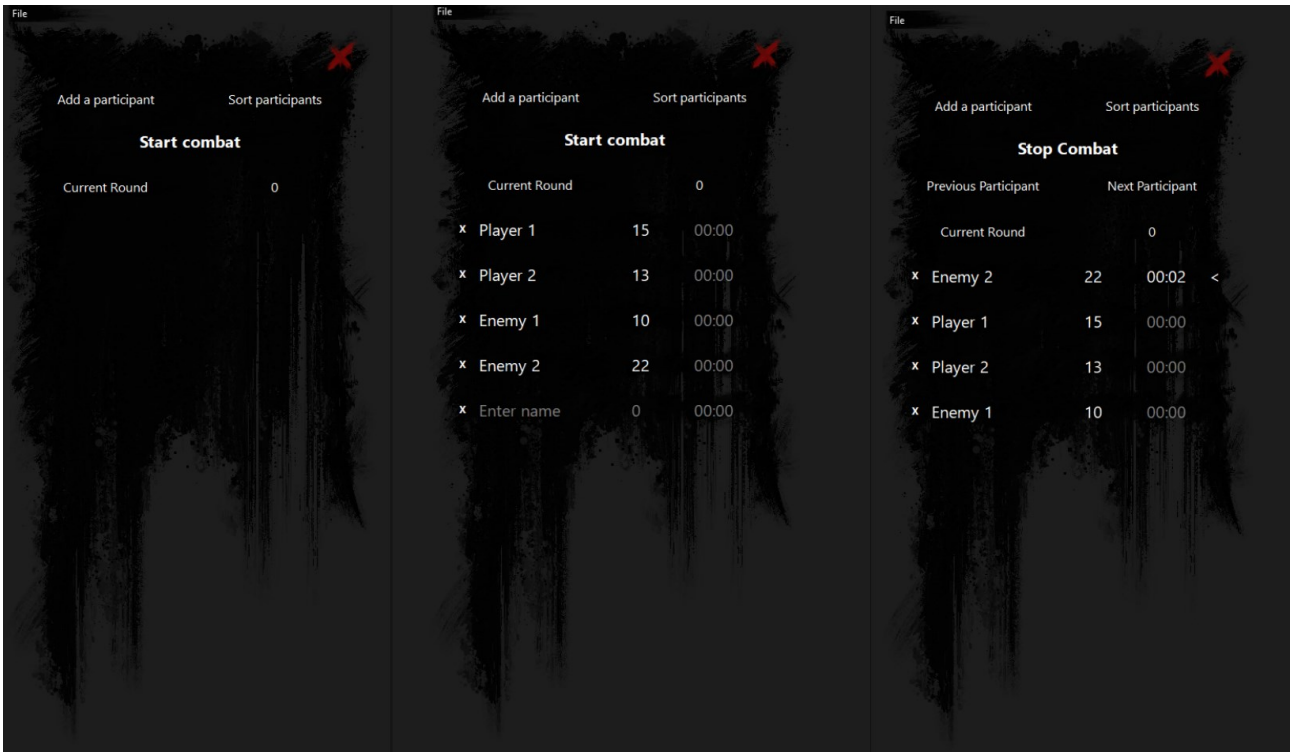

Figure 1. The GUI of the application in different stages.

The application features 4 different buttons and a menubar. The button marked with a red cross closes the application. “Add participant” button adds a new row to the combat list, which has a delete button, a label field for name, a label field for initiative, a label for timer and a label for current turn. “sort participants” sorts the list in descending order based on the initiative value of each row. Start combat marks the first member of the list with ‘<’ character, indicating that it is their turn. It also creates two new buttons called “previous participant” and “next participant”, these move the ‘<’ mark through the list in the combat order, circling through the list when the mark reaches the end. As the ‘<’ mark passes through the list’s last entry with “next participant”, the current round integer label is incremented by one. If the first entry of the list is passed with “previous participant” the label is decreased by one instead. The menu bar has two buttons, “save participant list” and “load participant list”, which let the user to store the current participant list as a JSON file to wherever they want to and the latter button lets the user to load a stored list from this file.

Though the program itself is not as heavy on the security aspect as some other excellent projects like password managers or authentication software, this program is related to memory management which was handled earlier in the course during the first weeks. Therefore, I claim that it is sufficiently categorized to fit the course theme.

2. How the program was implemented

The project has been created with the help of ChatGPT 4 and it has produced more than 70 percent of the code. The code is done in C++ and utilizes Qt framework for graphical user interface. The architecture, GUI design and graphics were created without the use of AI tools.

It was supposed to have other features which would have been related to secure programming, like encryption with the JSON files and the ability to run in the browser as a webapp, but I lacked the skill and time to do that, so this is a very slim feature wise in secure programming field.

3. Secure programming in the program

The OWASP Top Ten list was utilized to make sure that the program is secure. Unfortunately, the list had many security aspects which were not relevant for the program. Nevertheless, here is the list and what I discovered about my program's security aspects:

1. Broken Access Control: Does not affect my program, because it has no access control.

OWASP list security risk	Noticed risk	Discussion
Broken access control	Irrelevant, my program doesn't have access control.	Though the data is not secret, for education purposes it could be treated as such and access control could be implemented in the future.
Cryptographic failures	Irrelevant, my program doesn't have any cryptographic features.	This would be implemented with the access control as the potential passwords need to be hashed securely.
Injection	This is a big risk, the application takes user input and if that input is not validated, it could cause harm	
Insecure design	As the application has very few security elements, it can be considered insecure.	By adding access control and cryptography to passwords or perhaps participant JSON files, that would improve the secure design of the program.
Security misconfiguration	Irrelevant, my program doesn't currently use any type of security configurations.	The previously mentioned security aspects would require keeping an eye on this, they would need to be implemented correctly.
Vulnerable and outdated components	Low risk, the currently utilized framework of Qt is version 6.9.0 and therefore up to date, but it can still contain unknown vulnerabilities.	
Identification and authentication failures	Irrelevant, same as broken access control, there is no need for authentication.	
Software and data integrity failures	Medium risk, this is a possibility, as the data can be accessed by anyone who knows where the JSON file was stored and thus they can change it.	Though this won't cause any negative consequences just yet, it might in the future if the application's features are expanded.
Security logging and monitoring failures	Irrelevant, my program doesn't have logging or monitoring features.	

Server-side request forgery	Irrelevant, the program is a desktop application and has no internet features.	

Additional checks were made through Qt's documentation and their list of known vulnerabilities [2].

The Qt's documentation provides insight in to the risks of injection and memory overflow. My program's input fields which the user can edit, are implemented with QLineEdit class, which uses QString class to handle the text. This QString class is resistant to buffer overflow as it allocates heap memory dynamically. However to make sure that nothing unforeseen happens, I used QLineEdit's setMaxLength function to limit the possible input size to just 20 characters.

This also makes reading the JSON files safer as the QLineEdit will now truncate text that exceeds this 20 character limit, meaning that the fields in the JSON file cannot be used to cause memory overflow. However the JSON field can still be a potential risk, because even though it won't cause any problems with the QLineEdit field, it can move a problem downstream if that QLineEdit's information is later used somewhere else. In my case it is not, but if the program were to be expanded, this would have to be taken in to consideration.

```
// Participant's name
QLineEdit* nameEdit = new QLineEdit(rowWidget);
if (participant->name == "Enter name") { nameEdit->se
else { nameEdit->setText(participant->name); }
nameEdit->setPlaceholderText(participant->name);
nameEdit->setMaxLength(20);
```

Figure 2 How the QLineEdit is created for participant's name.

Similarly to the name field, the initiative score is also limited to only integers and with a valuerange from 0 to 100.

```
// Participant's initiative score
QLineEdit* iniEdit = new QLineEdit(rowWidget);
if (participant->ini == 0) { iniEdit->setPlaceholderText(QString
else { iniEdit->setText(QString::number(participant->ini)); }
iniEdit->setValidator(new QIntValidator(0, 100, rowWidget));
```

Figure 3 How the QLineEdit is created for the participant's initiative score.

Setting up these fields safely at the stage when the participant row is created, also has the added benefit that when the information is stored, it doesn't need to be checked separately as it will be stored exactly the way it is in the QLineEdit field.

The potential harmful actions were tested manually by hand but could benefit from automated tests.

4. Possible future plans

As the program's functionality is very thing, it did not have many features which could utilize the secure programming elements taught during the course, but it could be expanded with more time. These possible additions could be: encrypting the participants when they are stored in the JSON file, this way even though it does not contain any sensitive or secret information, it could be simulated. This encryption could then be analysed for vulnerabilities. Other option is to add browser functionality,

meaning that the program could be turned into a webapp, this would open another set of potential vulnerabilities, seen in the Google Gruyere exercises.

Finally some missing security features which I wanted to apply, were bleaching to the user inputs and JSON files, because it is possible that a user's supplied JSON file contains text which is not allowed. I discussed the potential harm of the JSON file earlier and for that reason bleaching or checking if the JSON file contains safe characters would be beneficial even in this stage where it cannot cause harm just yet.

5. References

[1] OWASP Foundation (2021) OWASP Top 10:2021 – The Ten Most Critical Web Application Security Risks. Available at: https://owasp.org/www-project-top-ten/ (Accessed: 7 May 2025).

[2] Qt Project (2025) List of known vulnerabilities in Qt products. Qt Wiki. Available at: https://wiki.qt.io/List_of_known_vulnerabilities_in_Qt_products (Accessed: 7 May 2025).

[3] The Qt Company. (2024) Qt 6.9.0 Documentation. Available at: https://doc.qt.io/qt-6/ (Accessed: 8 May 2025).