2025

Programming exercise report

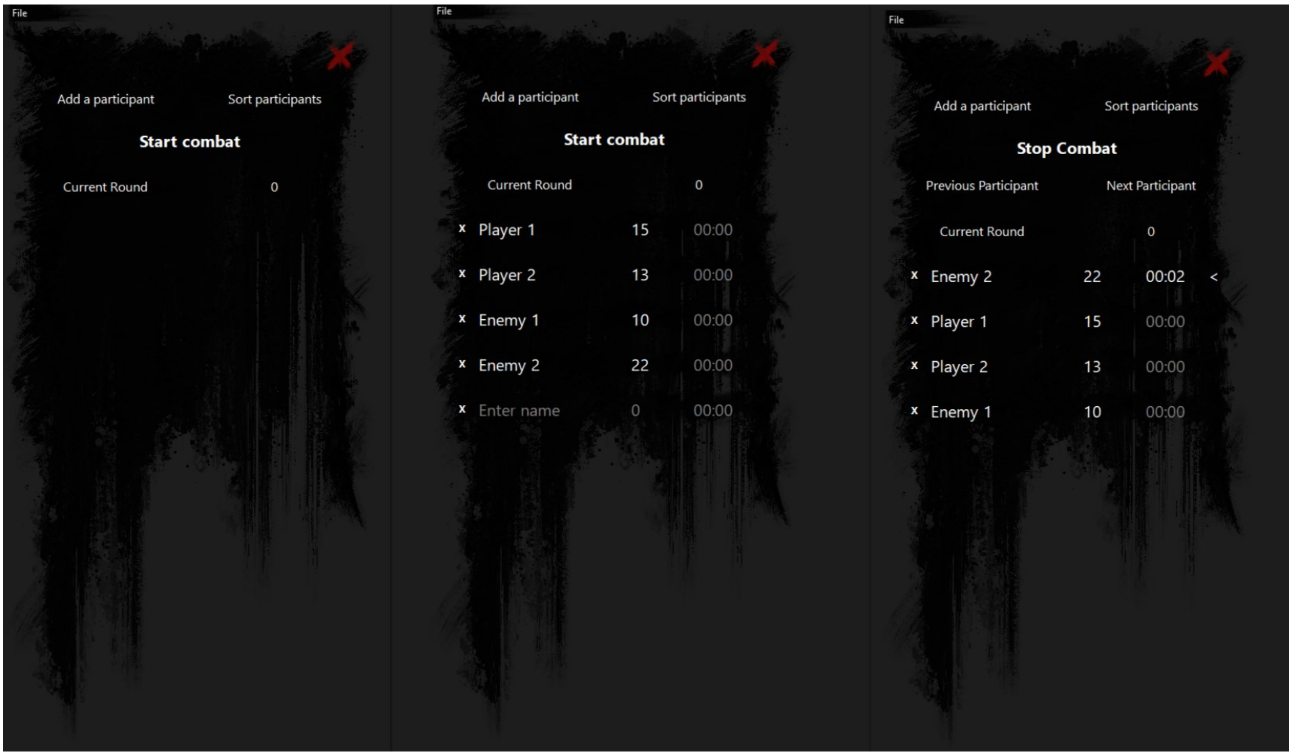
COMP.SEC.300-2024-2025-1 SECURE PROGRAMMING

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.



x	Player 1	15	00:00
x	Player 2	13	00:00
x	Enemy 1	10	00:00
x	Enemy 2	22	00:00
x	Enter name	0	00:00

x	Enemy 2	22	00:02	<
x	Player 1	15	00:00	
x	Player 2	13	00:00	
x	Enemy 1	10	00:00	

Image 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. 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.

ChatGPT was used to implement the functionality, designed in the application's architecture, meaning that it produced the code for individual functions. The application features around 1200 lines of code, out of which 200 is created with Qt designer extension for Visual Studio and of the remaining 1000 lines, approximately 75% is created with ChatGPT.

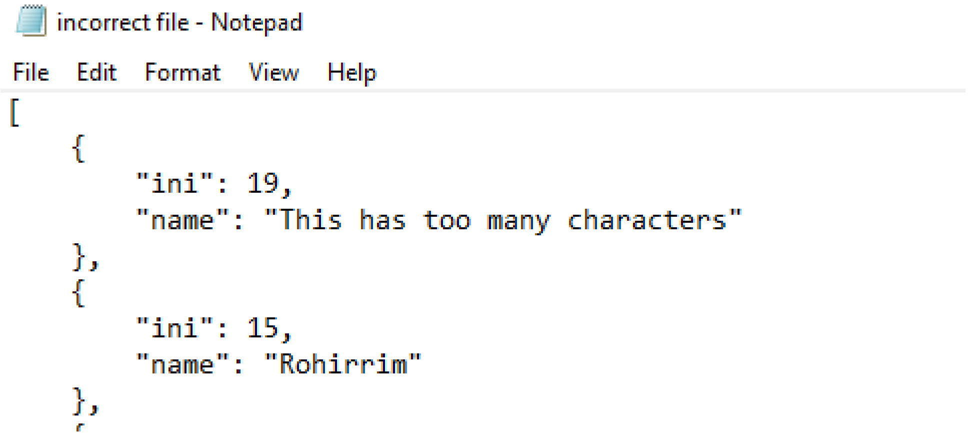
It was supposed to have other features which would have been related to secure programming, like encryption with the JSON files and the ability to run in the browser as a webapp, but I lacked the skill and time to do that, so this is a very slim feature wise in secure programming field.

3. How testing was done

The potential harmful actions were tested manually by hand but could benefit from automated tests. The manual tests first tried to seek problems in regular execution of the code, this means that pressing every button has the desired result and there were almost no problems with these tests except one. The tests which had unexpected results and how they were solved are listed here:

1. In the original sorting function, which was created by ChatGPT, there was a bug which caused the program to crash if trying to sort the list when it had participants. This was due to a problem with null pointers. When sorting, the function stored each row to a vector of participants, then cleared the list, however the vector store pointers to the real value, meaning that when the list was cleared, it also caused the pointers to point at nothing. Now when the list would be built again in the correctly sorted order, trying to access the vector's values crashed the program.
    - This was fixed by storing the participant values of each row as a copy of the original, instead of being a reference to the original.
    - As learned on the course, when program crashes it can generate a core dump, which can reveal private information like user data, memory or credentials, this means that fixing this bug was more than just making the application work as intended, it is also now more secure.
2. As the participant rows are the core of this application, the input fields were thoroughly tested with manual inputs of different kinds. These tests were trying to type too many characters in to the QLineEdit fields, trying to copy too many characters in to them and trying to run html code on them. No problems were found and after reading through Qt's documentation, it seems to be thanks to how the QLineEdit is implemented, more on that in chapter 4 [3].
3. The JSON file generated by the code was tested with different values. As the file is just a regular text file, it can be easily accessed by anyone who has access to the computer and thus the file can be edited to contain something unintended. An example of this is image 2 which shows a

name that exceeds a 20-character limit, which was set in to the QLineEdit. No problems arose from the JSON file containing too many characters or even when containing rich text.



```
[
    {
        "ini": 19,
        "name": "This has too many characters"
    },
    {
        "ini": 15,
        "name": "Rohirrim"
    },
```

Image 2 How JSON file looks, with a name that has more than 20 characters.

4. The windows executable file was also tested manually on different environments, both on my own Windows 10 machine which has all the necessary tools such as qt 6.9.0 and on a friend's Windows 10 machine which doesn't have anything related to programming. It was deemed to work without any problems in both environments.

4. 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.

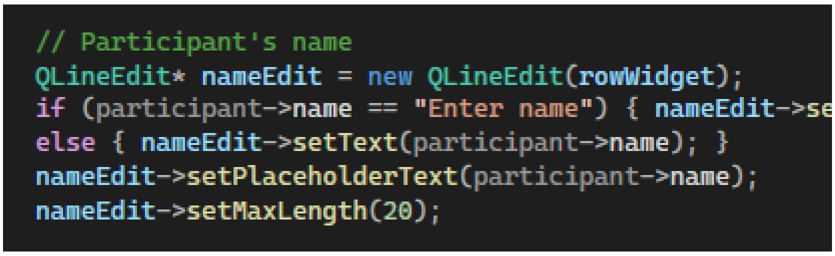
OWASP list security risk	Noticed risk	Discussion
Broken access control	Irrelevant, my program doesn't have access control.	Though the data is not secret, for education purposes it could be treated as such, and access control could be implemented in the future.
Cryptographic failures	Irrelevant, my program 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].

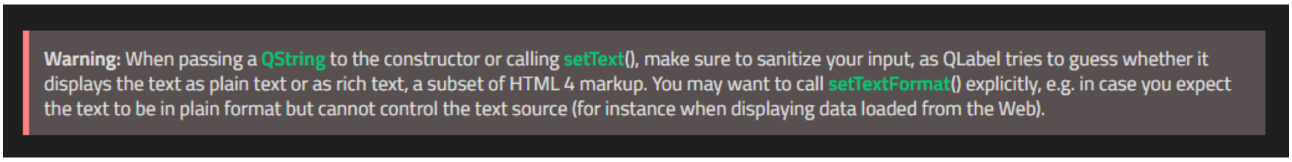
The Qt’s documentation provides insight in to the risks of injection and memory overflow. My program’s input fields which the user can edit, are implemented with QLineEdit class, which uses QString class to handle the text. This QString class is resistant to buffer overflow as it allocates heap memory 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 through enriched text, like html code. The QLineEdit itself won’t cause problems, because it treats all of it’s text as plain text, but it can move the 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. An example of this would be QLabel which can display the text as rich text.



```
// 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);
```

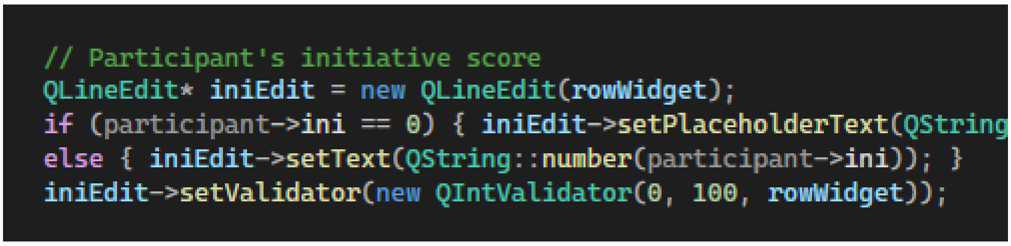
Image 3 How the QLineEdit is created for participant’s name.



Warning: When passing a QString to the constructor or calling setText(), make sure to sanitize your input, as QLabel tries to guess whether it displays the text as plain text or as rich text, a subset of HTML 4 markup. You may want to call setTextFormat() explicitly, e.g. in case you expect the text to be in plain format but cannot control the text source (for instance when displaying data loaded from the Web).

Image 4 A warning on Qt's QLabel Class' documentation.

Similarly to the name field, the initiative score is also limited to only integers and with a value range 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));
```

Image 5 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.

5. 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:

1. 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.
2. 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. An immediate problem is that this has the potential to trigger html code as seen in the exercises. Thanks to Qt's own documentation I also know that the QLineEdit fields are safe regarding this as they treat all the text shown as plain text.
3. Add an authentication window before the application launches the main GUI. This would be an excellent security related feature and would require the use of salt and pepper with the hash of the user's password.

Finally a missing security feature which I wanted to apply, was bleaching the user inputs and JSON files, because it is possible that a user's supplied JSON file contains text which can become harmful in the future even though it can't do anything just yet thanks to QLineEdit's secure implementation. There is always room for growth and being aware of this potential risk now is very beneficial.

7. 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).