# Introduction

The game we are implementing for our ISA 681 Secure Software Design final project is Blokus. This project was the joint effort of Pablo Guerra and Charles Tucker. This project is due on Sunday 5/13/2023. Our project utilizes a framework called Flutter which is described briefly below.

## Flutter

Flutter is a DART framework developed by Google for the creation of secure web applications. Its flexible design allows it to be compiled and deployed to several environments depending on the needs of the user. Many modules exist to make the creation and use of secure code easier to manage.

# Design/Architecture

# Installation Instructions

# Operating Instructions

# Game Rules

Blokus is a game about placing colored blocks on a gameboard. Play begins with blue, followed by yellow, then red, and finally green. Pieces must be placed such that they are corner to corner with a piece of the same color and do not overlap any other piece of any color. Additionally, you may not touch edge-to-edge with any of your other pieces. At the end of the game every individual square you did not place counts as -1 point, i.e. a block with 4 total squares in it counts as -4 points if not placed. If a player places all their blocks they are awarded +15 points. At the end of the game, the player with the highest number of points wins.

# Why It Is Secure

Several techniques were utilized to ensure the code is as reasonably secure as we could make it. This includes peer-reviewing all code that was written, performing code analysis, and reviewing common vulnerabilities to ensure that we are not susceptible to them. The process and purpose of these security mitigations will be described below. Additionally, the code is written using a DART framework called Flutter, which assists in managing the security of the code.

## Flutter

Flutter, as mentioned above, is a DART framework designed by Google for the development of secure web applications. We utilized its built-in capabilities as well as a few additional modules to ensure that the security of the backend server was properly managed. Before development started, we made sure to research the security posture and history of Flutter. Since its release in 2017 there have been a total of 6 CVEs found affecting the Flutter framework [2]. Of those 6, a total of 4 were of Medium or Low severity. In addition, all 6 CVEs have since been patched and are no longer exploitable. Since we are running the latest version of Flutter, we are confident that the current existing CVEs for this framework would not affect our code. Additionally, Flutter has seen widespread usage in mobile development, as well as other areas, tying the other most common framework at a 42% usage rate. Despite its wide usage, only a total of 6 CVEs has been found in the previous 6 years meaning we are confident that their security practices are solid and, judging by this historical data, are unlikely to cause a large issue in the future.

## Peer Review

Throughout the entire software development process, all code was reviewed, and tested by the other group members to ensure best practices were maintained and all code was secure. Many vulnerabilities introduced into code are caused when a programmer spends hours working on a project causing fatigue, and sometimes frustration, to result in less-than-perfect code being written. Peer reviewing allows for a second set of eyes to ensure the code that was written does exactly as it is intended to do. Much like proofreading an essay, rereading code, and having someone else double-check your work allows for most bugs to be discovered before further testing is even performed.

## Code Analysis

<Run code the static code analysis once complete>

## Common Vulnerabilities

Throughout the entire development of the code, we remained cognizant of common vulnerabilities which are found in web applications. We reviewed the OWASP 10 Ten for web applications [1]. Below is a list of the most common vulnerabilities in web applications according to OWASP and why we are confident that our code could not be comprised by them.

## Broken Access Control

In our web application, the only access control that is needed is control over whom a player can act as and what stats a player is allowed to see. In order to view a player’s stats or make moves on another player’s behalf one would need to sign in as that user. While we cannot stop users from generating bad passwords, on our end passwords are stored as salted hashes to ensure that an attacker would have a hard time stealing credentials even if they were to gain access to the password file. Additionally, <talk about how accounts work and why this is not a concern once implemented>

## Cryptographic Failures

The cryptographic failures vulnerability occurs when any form of the cryptography of the application fails to do the job it is expected to do. As mentioned before, Flutter handles a large part of our encryption. Since Flutter itself has no known vulnerabilities and is run by a reputable organization we can be fairly confident that our application does not contain any cryptographic-related vulnerabilities. Additionally, anytime sensitive data is stored or transmitted, mainly passwords, we utilize appropriate cryptographically secure hash or Flutter’s capabilities to ensure that the information is secure.

## Injection

The overall web app is broken into two sections, the first is the game itself and the second is the login pages. The game itself does not have any place for a user to input data, instead, it requires dragging and dropping objects. This means that the main game itself does not have an opening for a user to perform any kind of injection attack.

The login page does allow the user to input a username and password. For the creation of accounts, we set the requirements for usernames and passwords to be <talk about limitations of username and password here>. This allowed us to maintain strong passwords, but not open ourselves up to attacks based on these special characters. We utilized regex expressions to ensure that no forbidden characters were included in the input fields.

## Insecure Design

Our web application contains very few components, namely the server and the clients. All communication between these two components is performed over HTTPS. This ensures that the communication is secured and not susceptible to eavesdropping or integrity concerns. Also, care was taken to ensure that only necessary information is passed between the two components. This allows for greater certainty with regard to the privacy of the data.

## Security Misconfigurations

There are very few configurations implemented in this web application. Even still we have ensured that best practices are followed throughout the development of the application. There are no default passwords, no unnecessary services or bloat from unused code, and all software is patched and at its most up to date.

## Vulnerable and Outdated Components

This type of vulnerability relates to out-of-date languages, modules, or other parts of the overall system. Since this is a relatively small project, we have been able to verify that all components of this code are at their most up-to-date and that none of these versions contain any known CVEs or exploits that need to be mitigated.

## Identification and Authentication Failures

<Once this is implemented go back here and add how we did it securely>

## Software and Data Integrity Failures

This vulnerability occurs when software is deployed with proper verification of the underlying modules or code and without proper testing. A common example is the use of CICD pipelines which may automatically update modules without a human in the loop resulting in unexpected code changes. Since this is a relatively new project, all modules, and the framework itself were designed from the group up to the most updated version of the code. Additionally, we have no automation in our development pipeline meaning any changes would require manually updating the code. Given this, we are confident that all module and code changes will be verified by a human before being integrated into the final code.

## Security Logging and Monitoring Failures

<talk about any logging and monitoring all we need to implement is trail of moves>

## Server-Side Request Forgery

<disable http redirections if possible and talk about that here>

## Conclusion

Since we have done our due diligence in reviewing the framework itself and the modules we have chosen to implement, we believe that our decision to go with a framework will not create a security risk. Additionally, given the practices we have chosen to implement, and our diligence in ensuring the more common web app vulnerabilities are not present. We can be reasonably confident that our code is well-designed and as secure as it could be.

# References

1. [OWASP Top Ten | OWASP Foundation](https://owasp.org/www-project-top-ten/)
2. [NVD - Results (nist.gov)](https://nvd.nist.gov/vuln/search/results?adv_search=true&isCpeNameSearch=true&query=cpe%3A2.3%3Aa%3Adart%3Adart_software_development_kit%3A-%3A*%3A*%3A*%3A*%3A*%3A*%3A*)