SWE 681 (Secure Software Design and Programming)

Dr. David Wheeler

Project Report

Pente

December 13, 2019

Hamza Mughal & Nabil Darwich

Introduction

Pente is a strategy board game which can be played by two or more players. Pente is based off the Japanese game ninuki-renju (which itself is a variant of the board game Go). Our project revolves around a web application that allows users to be able to play Pente through the web browser against other players. Our version of Pente supports only two players in any given game.

Users can do the following when they visit the web page:

1. Create an account
2. Log in with their credentials
3. Create a new game with a unique game ID
4. Join games that currently need players
5. See statistics (win/loss/tie ratios) of all players
6. See move history of all completed games
7. See their currently pending games (allowing them to rejoin if they get disconnected)
8. Log out

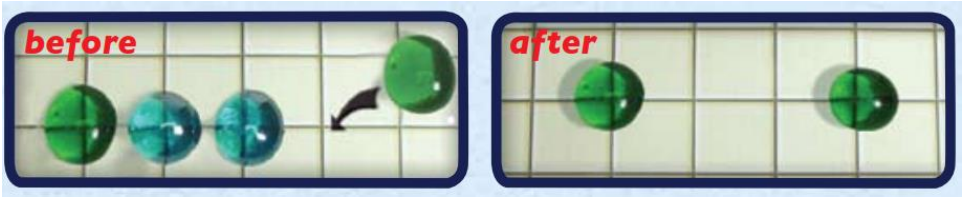
The main focus of the project was to develop a secure web application. For example, preventing players from actively cheating against their opponent to secure a win. The three major security requirements that have been implemented are confidentiality, integrity, and availability.

Game Rules

There are only two ways to win at Pente. The first is to have 5 stones in a row either horizontally, vertically, or diagonally. The second is to capture at least 5 pairs of the opponent's stones.

The game starts with an empty 19x19 board. The starting player, white, places their stone down anywhere. Players then alternate turns placing pieces at empty locations. As soon as a piece is placed, the other player starts their turn. Pieces cannot be moved once they are placed on the board.

Whenever the opposing player has two adjacent stones (and only two), this pair can be captured. Pairs are only able to be captured if and only if one stone is placed at each end of the opposing player's pair. For example:



Captures can be made diagonally, horizontally, and vertically. Captured pairs are removed from the board.

Any intersection that was previously captured may still have a piece placed even though it may have been occupied previously.

Multiple pieces can be captured as well as follows:

Design and Architecture

The project is broken into different folders and subfolders as follows:

```
public_html/
    database/
        database.json (database of user credentials)
    css/
        pente.css
    js/
        createAccount.js
        home.js
        joinGame.js
        leaderboards.js
        pente.js
        penteGame.js
        gameHistory.js
        completeGame.js
    public/
        pages/
            pente/
                createAccount.html
                game.html
                home.html
                joinGame.html
                leaderboards.html
                gameHhistory.html
                completeGame.html
        routes/
            pente.js
    app.js
```

Our web application was designed primarily using NodeJS, ExpressJS, and JavaScript.

NodeJS

NodeJS is an open source JavaScript runtime environment that executes JavaScript code outside the web browser.

ExpressJS

ExpressJS is a web application framework specifically for NodeJS. ExpressJS makes developing web applications more convenient. It is based off the NodeJS module connect which in turn uses the HTTP module. With ExpressJS, we are easily able to serve files to a client, define routes based off HTTP methods, and include various middleware modules that are necessary to complete any tasks.

JavaScript

JavaScript is a programming language that allows us implement logic on web pages such as updating a timer, sending user information for authentication, and playing an online two player game.

Major Components

Database

Rather than using commercially available DBMS software (such as SQL server, MongoDB etc), we opted to simply represent our database as JavaScript objects in the server code. This eliminates the need to connect to an external database to be able to store and retrieve data. We create and store a local copy of users that have registered, so that they do not have to create a new account if the server restarts.

Socket.io

Socket.io is a JavaScript library which enables real time web communication between web clients and servers. There are two parts to Socket.io – one runs on the client side and the other on the server side (using the socketio library for NodeJS).

Major Files

app.js

This file is basically the brains of the whole operation. It contains databases that map players to games, games to players, and it contains all of the logic necessary for verifying the game state on the server. This helps ensure no cheating is possible, as the rules are enforced by the server. This also helps a user reconnect to an active game if they get disconnected. This file serves as a main source of communicating between two clients in the game ensuring both of their game states are consistent.

routes/pente.js

This file serves primarily to route requests to the various pages of the application only if they are authenticated (meaning they have an active session via logging in). Additionally, this file handles logic for creating user accounts, authenticating users and giving them a valid session, and retrieving data from databases (such as in the case to view move history of past completed games).

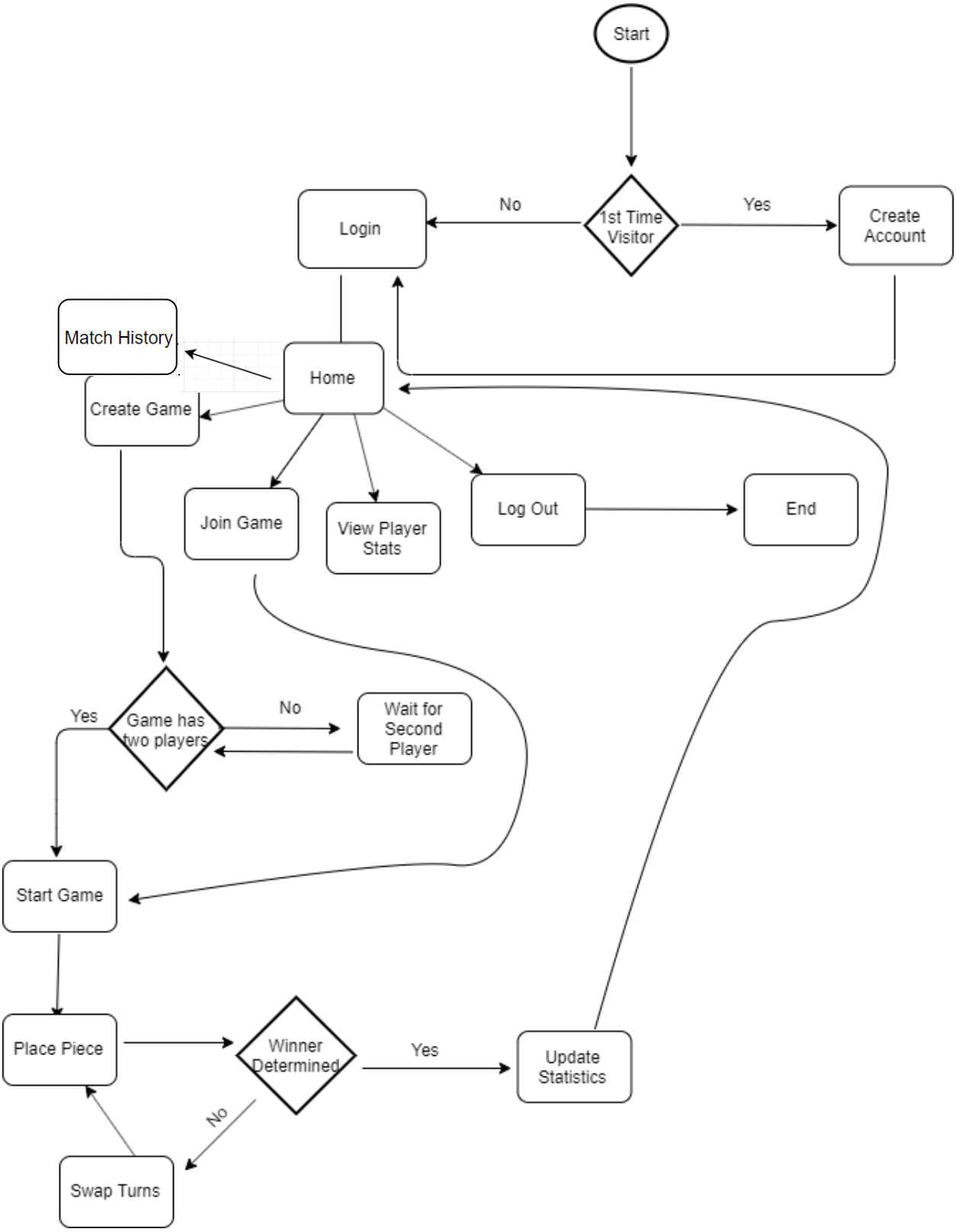
Installation Instructions

1. Install node package manager (npm) and NodeJS at this link https://www.npmjs.com/get-npm. It is required to be able to successfully install all the required modules (NodeJS will automatically be installed along with npm). To make sure it has been installed properly, you can type node -v and npm -v in the terminal.
2. Download the repository from Github - https://github.com/ndarwich/nabildarwich.com
3. Once the repository has been installed, at the root type npm install. This will install all the required dependencies (this may take a few minutes).
    a. Go to penteGame.js and change the io.connect string to http://localhost:3002. Please note it may be nearly impossible to actually run the game locally against yourself as the sessions will be the same (thus not allowing you to play against yourself locally). We have a live version of the game available on http://nabild.com/pente which allows you to play against yourself with different accounts in two different browsers.
4. Start the server by typing either gulp or npm start
    a. The page will automatically open to the home page on localhost:3000, please go to http://localhost:3000/pente to see the project locally.

Live Version of the game available at http://nabild.com/pente

Demo - https://www.youtube.com/watch?v=AucVpNuPdM0

Operating Instructions



Why we believe our software is secure

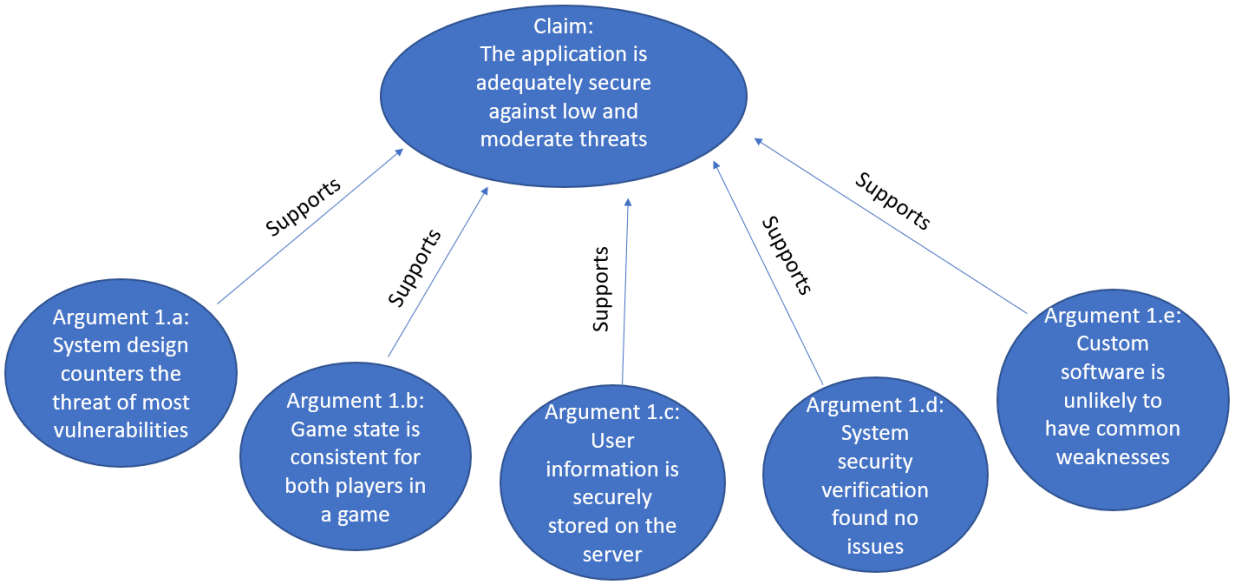
The security requirements we have met were the following:

1. Confidentiality – Every move that is done in a game is logged by the server, and these logs are not available to those not in the game until after the game has completed.
2. Integrity – Only players that are in the game can make a move, and those moves can only be syntactically correct. This is verified by the server to ensure the integrity of the game. Player statistics are also appropriately updated on a win/loss/tie.
3. Availability – The game does not stall forever if a player does not make a move. Players are given at most 1 minute to make their move otherwise they forfeit the game.

Assurance Case

Claim: The application is adequately secure against low and moderate threats.

Argument 1.a – System design counters the threat of most vulnerabilities
Argument 1.b – Game state is consistent for both players in a game
Argument 1.c – User information is securely stored on the server
Argument 1.d – System security verification found no issues
Argument 1.e – Custom software is unlikely to have common weaknesses



High level overview of the Assurance Case

Argument 1.a: System design counters the threat of most vulnerabilities

Argument 1.a.i - All un-trusted inputs id'd and checked by strict white-lists

Evidence – Regular expressions are used to validate un trusted user input. The primary source of un trusted user input comes from the create account page. There are client-side checks, however this can easily be modified and removed by the user, so the same checks occur on the server. The user inputs are checked with a whitelist to ensure they meet the criteria for usernames and passwords (meaning that they cannot input any non-alphanumeric characters).

Argument 1.a.ii – Use of Dictionaries to represent the database prevents SQL attacks

Evidence – Rather than using a DBMS for our database, using simple JavaScript dictionaries with keys and values makes it impossible for any sort of SQL injection to occur. We do not have to query out to an external database to retrieve any sort of data as it is already stored on the server. This additionally eliminates the need to prepare any sort of SQL statement as we can simply query our database dictionary with a key.

Argument 1.a.iii – Minimal attack surface

Evidence – The only source of potential input from the user occurs when creating an account, logging in, creating a game, and playing their piece during a game. No where else is a user allowed to enter any input. Additionally, unauthorized users do not have access to the game's web pages if they are not authorized/logged in (thus implementing defense in depth). Anytime a user wants to access a page, the first thing that is checked is whether or not if they have an active session. If they do not, they get redirected thus allowing us to eliminate the risk of any potential unauthorized user gaining access.

Argument 1.a.iv – Client Server Architecture

Evidence – The server checking for validity against all inputs provides the strongest type of security, allowing only authorized users access to the web application. Client-side code can be easily manipulated by a user and is not reasonable grounds for proving that the application can provide ample security. Server-side security is implemented to provide additional security counter measures against malicious users.

Argument 1.b - Game state is consistent for both players in a game

Argument 1.b.i – Game state is controlled by the server

Evidence – The current state of the game is stored and administered by the server. Anytime a new game is created, a unique game ID is created which maps to that game session. Users join a game by entering the game ID, which effectively registers them for the game session. The server additionally stores the whole board (stored as an array of characters) to the game dictionary. Anytime a player makes a move, it is sent to the server which logs the input and subsequently sends the input to the opposing player where their board gets updated. The server also logs every input made on the board by each player. The server is also responsible for checking for invalid input (done by checking against the board stored on the server for the game ID), The server additionally flags whose turn it is and effectively checks that the authorized user, whose turn it was, made the move when it was their turn. When the server detects that a player has won, it notifies both players that a player has won. The state of the game is entirely controlled by the server and prevents users from actively cheating.

Argument 1.b.ii – Game timer to promote availability

Evidence – A game timer is associated with every active game. As soon as a game starts, the timer starts. Because the server also keeps track of whose turn it is, if a player has not made a move after one minute the server notifies all clients in the game that the game is over because the other player did not make a move within a minute. Players need to rejoin the game within their turn time otherwise they forfeit.

Argument 1.c - User information is securely stored on the server

Argument 1.c.i – Salted hashes to store user passwords

Evidence – Before storing a newly created username and password in the database, we salt and then hash the password. The crypto module is used to generate random bytes which are used as the salt for the password. Afterwards, the salt is used in junction with sha512 to generate a hash. Once the hash is generated, the salt and hash are stored as the value for the username in the database. Passwords are never stored in plaintext and are thoroughly salted and hashed. If, somehow, an attacker was to gain access to the database they would not be able to easily guess the passwords as they have been encrypted

Argument 1.c.ii – User authentication

Example – Whenever a user logs in, the stored salt from when they created an account is retrieved. That salt is used to encrypt the password they entered to log in with, and if there is a match then the user is successfully authenticated and granted access. If not, they get sent a message letting them know either their username or password is incorrect. We chose not to send back that their password was incorrect because we would like the attacker to not be able to know if the username they entered was correct or not (this adds more complexity to their attack). Additionally the input that is entered for the username is checked against a whitelist in the server.

Argument 1.d - System security verification found no issues

Argument 1.d.i – Dynamic Analysis Tool Issue Fixed

The dynamic analysis tool we ran on our web application was OWASP ZAP. Initially it had found a problem with web browser XSS protection not being enabled. To fix this vulnerability, we modified the web server's apache2.conf file to add the following:

<IfModule mod_headers.c>

Header set X-XSS-Protection "1;modeblock"

</IfModule>

Another issue found was that X-Frame option headers were not set. To fix this issue, we used the helmet module in NodeJS to set the X-Frame-Options header. Additionally, on the server, we added

Header set X-Frame-Options "sameorigin"

Argument 1.d.ii – Static Analysis Tool Found No Vulnerabilities/Issues

The static analysis tool used to analysis our JavaScript code was ESLint. ESLint did not find any vulnerabilities/issues when it was ran against our source code.

Argument 1.e - Custom software is unlikely to have common weaknesses

Argument 1.e.i – No Injection Flaws Possible

Evidence – An injection, such as an SQL injection, results from failing to filter untrusted user input and then passing that to an SQL statement. Our application does not use any external databases, rather a JavaScript dictionary (see 1.a for more discussion). The dictionary can never receive an injection attack against it as keys are simply unique IDs in our case.

Argument 1.e.ii – No Broken Authentication

Some web applications either do not properly encrypt stored passwords, or make session IDs very predictable. Our web application appropriately salts each password and subsequently hashes it before storing it locally thus making it very difficult for someone to do a dictionary attack. To generate a unique, hard to guess session id for a session, we use the uuid/v4 module to generate a universally unique identifier (uuid) which is a 128 bit number. Because it is assumed that the uuid is universally unique, there is no chance another user will have the same uuid (and thus the same session).

Argument 1.e.iii – No Broken Access Control

Our application has the bare minimum amount of access that is needed for our project. Users are restricted to simply creating games, joining games, and viewing statistics of users/past games. Anytime someone tries to access a game they are not in(to potentially modify the outcome of that game), the default is to deny. No user can modify the contents of another user (their password/statistics etc) as they will not and never will have access to such underlying structures within our project.

Argument 1.e.iv – No Components Used Have Known Vulnerabilities

None of the packages used with our project have any known vulnerabilities. Those packages that did have vulnerabilities were due to the version of the package being not up to date. This is easily solvable by using npm to update packages to their latest version.