HazzardHulen.io

Thomas A. Lemqvist
thlem14@student.sdu.dk

January 7, 2017

Contents

Acknowledgements

This paper exceeds the five page limit, due to the usage of images and code snippets, If they were removed, it will abide the limit.

1 Introduction

This section introduces the application developed in the TEC course.

Together with a group of classmates, I created a Black Jack web-application, nicknamed Hazard-Hulen.io, where users can "sit" at the table and bet game money.

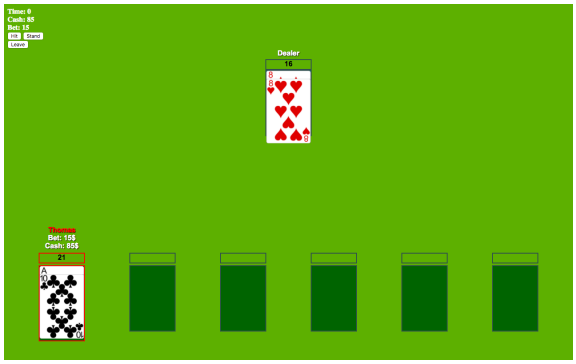


Figure 1: Singleplayer in the Black Jack web-App

Figure 1 shows the black jack table with one player, and the dealer AI, who makes decisions based on the score of his cards, to follow some common black jack dealer rules.

The game also scales to multiple players, currently only up to 6 players, due to the lack of table space. This could be changed, by introducing multiple tables to the system.

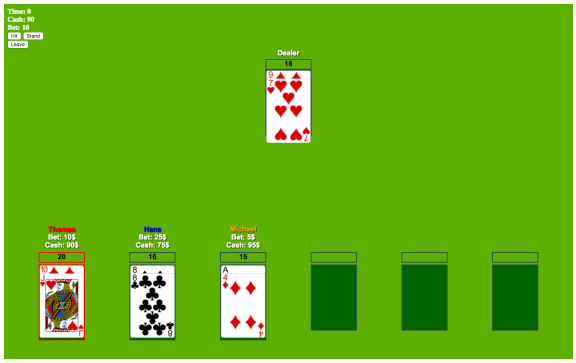


Figure 2: Multiplayer in the Black Jack web-App

Figure 2 shows three players sitting at the table. The player whose currently taking his turn is highlighted with a red edge around his score and cards.

Instructions upon how this can be run can be found in appendix A

2 Technologies

This section introduces technologies, which was chosen to work with.

2.1 TypeScript

TypeScript is a superset of JavaScript, and adds a more Object Oriented (OO) abstraction to JavaScript. It does this, by allowing the developer to write in a OO style, with interfaces, classes etc.
One of TypeScripts core features, is the ability to specify types for variables (number, string,

and boolean). This makes it easier for developers to avoid errors, where strings are inserted into numbers.
The TypeScript file can be compiled into plain JavaScript, which can then be deployed [1].

It can be implemented to easily configure the object structure of the system, in the case of HazardHulen.IO, it can be used to model the Table and Player objects.

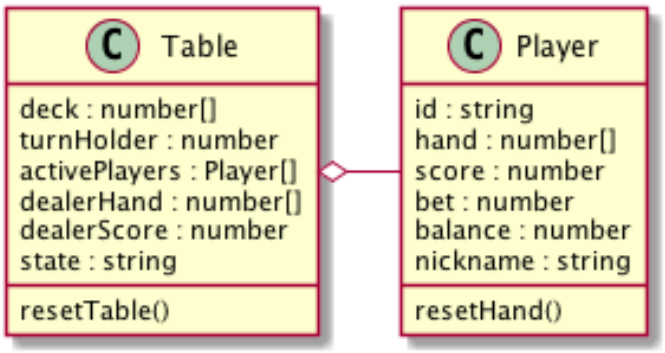


Figure 3: Class Diagram for HazardHulen.IO

2.2 Cros Site Scripting Protection

Cross Site Scripting (xss) is the injection of client side scripts into other clients, this can be done in comments or names, really anything which shows on other clients as embedded text.
it works by entering a `<script>...</script>` tag into a comment, this will be read by other clients as part of the websites own scripts and executed [2].

It can easily be prevented, simply by escaping the significant xss characters, in any input fields which shows text to other clients [3].

3 Implementation

This section describes the implementation of the technologies mentioned in section 2

3.1 Implementation og TypeScript

Because TypeScript (ts) is a superset of JavaScript (js), it is actually really easy to transfer the functionality from js to ts.

I have reimplemented the server in TypeScript.

I started by changing the file extension from *app.js* to *app.ts*, which is the TypeScript extension.

I then opened the a command line, and entered the `$ tsc -w app.ts` command, which compiles the app.ts file, each time it is changed. (-w = watchmode, which compiles on save)

```
class Table {
    deck : number[];
    turnHolder : number;
    activePlayers : Player[];
    dealerHand : number[];
    dealerScore : number;
    state : string;

    constructor(){
        this.activePlayers = [];
        this.resetTable();
    }

    public resetTable() {
        this.deck = _.shuffle(_.range(1, 52));
        this.dealerHand = [];
        this.dealerScore = 0;
        this.state = 'idle';
        this.turnHolder = 0;
    }

}
```

Code Snippet 1: TypeScript implementation of the table Object

I then implemented the table object, shown in Code Snippet 1. It has specified types on its variables.

```
var Table = (function () {
    function Table() {
        this.activePlayers = [];
        this.resetHands();
    }
    Table.prototype.resetHands = function () {
        this.deck = _.shuffle(_.range(1, 52));
        this.dealerHand = [];
        this.dealerScore = 0;
        this.state = 'idle';
        this.turnHolder = 0;
    };
    return Table;
}());
```

Code Snippet 2: the compiled Table

Code Snippet 2 shows the js code, which was generated form the ts code in Code Snippet 1. It is plain JavaScript, and can be executed to have the server running.

3.2 Implementation of Cross Site Scripting Protection

I started by identifying the places in the front end, which could be vulnerable to injection attacks.

And i found two places. when the player has to set his nickname, and when he has to bet, it is possible to inject code into these input fields.

I figured that i could just make a function which escaped the significant characters [3].

```
function validateInputForJs(input) {
    input = input.replace(/(&)/g,   "&amp");
    input = input.replace(/(<)/g,   "&lt");
    input = input.replace(/(>)/g,   "&gt");
    input = input.replace(/(")/g,   "&quot");
    input = input.replace(/(')/g,   "&#x27");
    input = input.replace(/(\/)/g,  "&#x2F");
    return input;
}
```

Code Snippet 3: Function which escapes significant XSS characters.

Code Snippet 3 shows the function which escapes the significant characters:[`&`, `<`, `>`, `"`, `'`, `/`], by locating them in the string by using regex [4], and then replacing them with their hex representation. This will make them be appear as if they didn't change, but they will not be executed as scripts anymore.

4 Comparison

This section analyses and discusses the impact of the implementation in section 3

4.1 TypeScript

I decided to test the responsiveness of the "new" server, to see if the TypeScript implementation had had an impact on this.

I found a tool called loadtest [5], which can be used to test the responsiveness of a Node server. It does this by setting up concurrent sessions, which makes requests to the specified target, it is very similar to a "controlled" dos attack[1].

```
$ loadtest -t 60 -c 20 --rps 300 http://localhost:13337
```

Code Snippet 4: The command to execute the loadtest.
`-t = time -c = number of sessions --rps = requests/second`

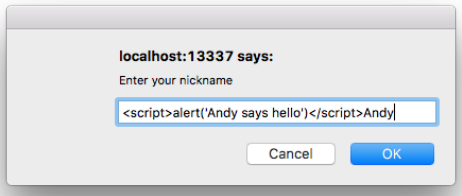
Code Snippet 4 shows the command I used to test the server responsiveness. I tested for 60 seconds, with 20 concurrent sessions, and 300 requests per second.

The tool then returns information about the responsiveness of the server, and gives a mean latency.

4.2 Cros Site Scripting

I tested this, on an older version of the application, by setting my nickname as `<script>alert('Andy says hello')</script>Andy`

[1] denial of service attack https://en.wikipedia.org/wiki/Denial-of-service_attack



This resulted in my nick name becomming Andy (see Figure 4), and a alert message popping up each time someone does anything. This is really annoying, but it is far from the worst thing one could do with xss, this example is very loud (and visual), it could potentially be used to download malware unto an unsuspecting user machine [6].

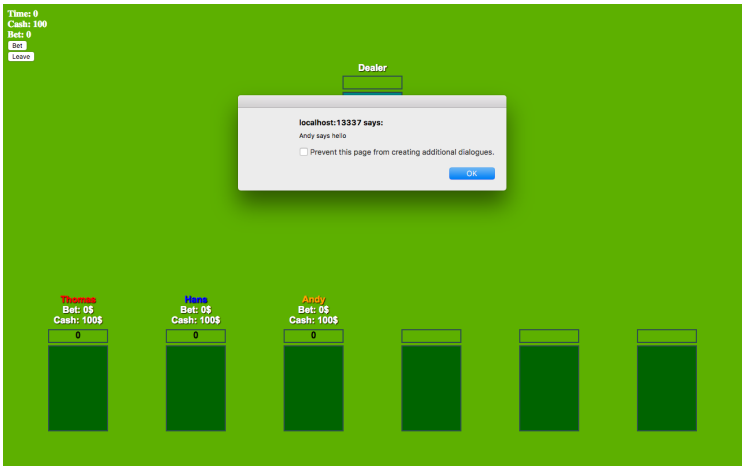


Figure 4: Image showing the dialog, as a result of the xss injection

I then did the exact same thing, on the version which implements the escape function.

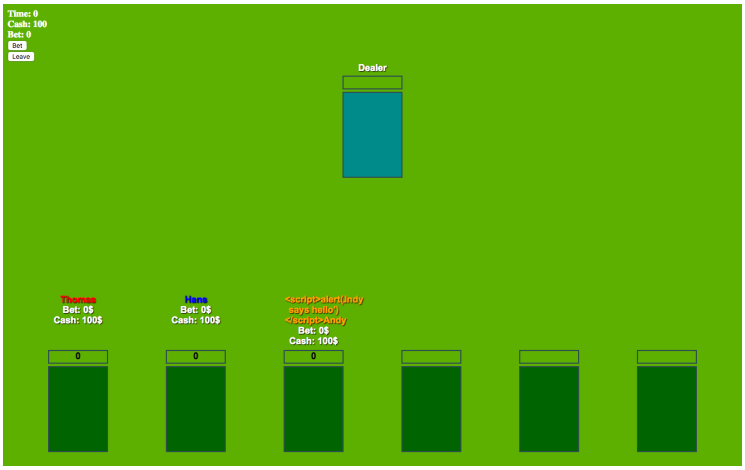


Figure 5: Image showing the failed xss injection

Because the characters has been escaped properly, they show up as they should (see Figure 5). internally in the server the string is converted into this:
`&ltscript&gtalert(&#x27Andy says hello&#x27)&lt&#x2Fscript&gtAndy` (this was copied from the console.)

5 Conclusion

This section concludes the

The Cross Site Scripting (xss) was successfully escaped, and has made it more inconvenient for attackers to inject JavaScript into the website.

References

[1] "Type script website," https://www.typescriptlang.org.

[2] Wikipedia.org, "Cross-site scripting," https://en.wikipedia.org/wiki/Cross-site_scripting.

[3] OWASP.org, "Cross site scripting prevention cheat sheet," https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[4] Wikipedia.org, "Regular expression," https://en.wikipedia.org/wiki/Regular_expression.

[5] Alexfernandez, "Loadtest github repository," https://github.com/alexfernandez/loadtest.

[6] OWASP.org, "Xss attack consequences," https://www.owasp.org/index.php/XSS#XSS_Attack_Consequences.

List of Figures

List of Code Snippets

Appendix A How to Run

To run the server, you need to have npm (it can be downloaded along with node here: `https://nodejs.org/en/download/`).

Run the following commands, in the server directory, to start the server:

```
$ npm install
```

Install bower by running this command, in the command line to install bower: (from: `https://bower.io`)
NPM might need root permissions to install bower as it is installed globally (-g).

```
$ npm install -g bower
```

To start the server, now you simply just has to run the following command.

```
$ npm start
```

Now the server should be up and running.

To connect a client to the server, go to `http://localhost:13337/` (or click the link)