DD2525 Language-Based Security

Security Implications of WebAssembly Sandboxing

Axel Lindeberg & Albin Winkelmann

The code is available at https://github.com/AxlLind/wasm-sandbox-demo

Contents

1 Introduction

Javascript runs the modern world wide web. According to W3Techs (2020), 95% of the top 10 million most popular websites use client-side JS. Each time you visit one of these sites you download megabytes of Javascript code, containing anything the website wants to run. On most websites Javascript is loaded from third-party sources as well via content delivery networks (CDNs). Your browser then blindly runs this code. While there are many security checks in place, like the single origin policy for example, the web is ridden with security vulnerabilities and multiple times per day you run potentially malicious JS. One of the most common vulnerabilities is XSS, repeatedly ending up on OWASP's top 10 web security vulnerabilities (OWASP Foundation (2020)). One of the issues making untrusted JS into such a sever vulnerability is that when JS is served the script always has access to everything JS usually has access to. There are no permissions given to certain scripts, or any way to only allow certain features. A third-party library could for example send a HTTP request or read cookies without you knowing or having to give explicit permission.

WebAssembly (WASM) is a brand new technology originally aimed at the web. There are many different motivations behind why WASM was developed but among them are security. See section 1.1 for more details on what WASM is, how it works, and the motivations behind it's creation. One of the security features of WASM is it's sandboxing functionality. What this entails is that when declaring a WASM module it does not have access to everything the browser can do. Instead you explicitly expose what external functions it can call. These could be bindings to HTTP requests for example. Unless such a binding is provided it is impossible for the WASM module to perform a HTTP request. These exposed functions along with the functions the module itself exports to your applications are the only points of contact between the WASM module and the outside world. As such WASM has a great security potential for the future of the web.

This report will cover the basics of what WebAssembly is and how it works. It will focus on the security implications of WASM's sandboxing functionality. Along with this report, a demo has been developed, demonstrating how WASM can be used for safe execution in the web. This demo lets users upload code which other users can run in their browser (to play tic-tac-toe). Using JS for this would be a sever XSS vulnerability but with WASM this can be completely safe.

1.1 Intro to WebAssembly

WebAssembly, abbreviated as WASM, is an open standard for a binary instruction format. It became an open standard and widely supported by major browsers in March 2017 making it very young in the world of the web. WASM is executed in a stack-based virtual machine (more on that in section 3) making it efficient, fast and safe. It is made trying to bring high-level languages like C, C++, Rust, and their performance to the web. As of writing this, Mozilla Firefox, Chrome, Safari and Edge have support for WASM 1.0 (WASM Group (2020)).

1.2 Why we chose WebAssembly

We chose WebAssembly in this project because it is a brand new and groundbreaking piece of technology within web which also promises a much safer execution environment. It is a promising new technology and we wanted to learn more about it. From a security perspective it is very interesting but also for many other reasons. It, for example, enables other languages than JS on the web and has the potential to speed up CPU intensive tasks in the browser like web-based emulators, WebGL renders, and much more. However, the focus of this report will be on it's security implications.

2 Goal

The overarching goal of this project is to learn more about WASM from a security perspective. By researching how WASM works and learning more about it's specification we want to learn more about how it can be used and what consequences it can have for the future of the web. We also want to demonstrate how WASM can affect the security of a website with it's sandboxing capabilities by showing how an otherwise completely unsafe action becomes safe when using a WASM-module instead of plain JS for third-party code.

Letting users upload code that other users execute in the client is a textbook security vulnerability. This exposes the users to XSS in the most trivial way, since the server explicitly serves user generated code to other users. We want to show how this can be done in a safe way using WebAssembly by putting it in an extreme sandbox environment with very limited endpoints exposed to the outside world.

3 Background

This section covers the basics of WebAssembly: how it works, the main motivations behind it's creation, and why it can be much faster than Javascript. Additionally, sandboxing as a security mechanism is described. Lastly, there is a section detailing how the sandboxing of WASM modules work as well as the effect that can have on the web ecosystem.

3.1 The basics of Web Assembly

WebAssembly is, as described by the WASM Working Group:

> [...] a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

In contrast to Javascript which is a high level programming language, WASM is more akin to an assembly language for a specific ISA than an actual programming language. The idea is generally not to write WASM by hand but to instead compile other languages into WASM. It is stored in a binary format with specific opcodes denoting different instructions. This is also a major difference

to Javascript, which is sent as plain text which is directly interpreted by a JS interpreter.

However, WASM differs quite a bit from regular assembly like x86 or MIPS. The WASM instruction set targets a stack-based virtual machine that executes WebAssembly instructions (WebAssembly Working Group (2020)). Unlike other assembly languages, WASM is typed. It does not operate on any high level types. Instead, the only types the WASM VM knows are $i32, i64, f32, f64$ and all instructions operate on these types. This, however, can make interoperability with WASM modules relatively complex, since the VM itself is very low level. The executor and language that is compiled to WASM needs to agree on the low level memory representation of objects before they can share them along the WASM border. Today the ecosystem is not very mature but proposals like WebAssembly Interface Types aim to make this much easier. This proposal defines a standard for how to represent types like strings over the WASM border, so that all languages that target WASM will represent them the same way.

Another difference to regular assembly like x86 is what a WASM file contains. It is not just instructions and static data. Among other things, a WASM file contains the following:

- A function table. A major difference from regular binary files is that each function is numbered, starting at zero, instead of simply existing in a place in memory which the machine jumps to. This, along with it being a stack-based VM eliminates many memory related vulnerabilities like overwriting the return pointer for example. Functions can also be marked as exported. This means they are visible to the executor of the WASM module and callable by them. This is the first of only three ways for the module to interact with the outside environment.

- A list of imported functions, along with their signatures. These are named functions that the executor of the WASM module has to supply an implementation of. This enables more advanced functionality like for example system calls, or DOM-manipulation in the browser but the outside environment has to then supply this functionality. This is second of three ways for the module to interact with the outside environment.

- Instructions making up each function definition. This is similar to other assembly formats, containing for example numeric instructions like *add* or *mul*, and instructions to call other functions etc. All instructions, however, operate on specific types.

- Sections describing the memory section of the module. Unlike execution environments like x86 processors for example, the instructions and data memory are fundamentally separated. This prevents code from changing any instructions at run time, a common source of vulnerabilities. This memory region is exported and shared with the outside environment, i.e both can read/write to it. This is the third and last way for the module to interact with the outside environment.

WebAssembly was primarily intended for the web but has grown to be a general execution environment facilitating interoperability between all programming languages and computing platforms. There were many different things

that motivated the creation of WASM. Among the most important was creating a faster and more secure execution environment than Javascript in the browser.

3.2 Why WebAssembly is fast

Unlike JS, which is interpreted and then executed with just-in-time (JIT) compilation, in the browser WASM is compiled once and *then* executed. Since WASM is only compiled once and then optimized it is naturally faster in many ways than JS. Both in start-up time, since the binary format is much faster to parse, and execution time. To help with visualizing this concept consider figure 1.

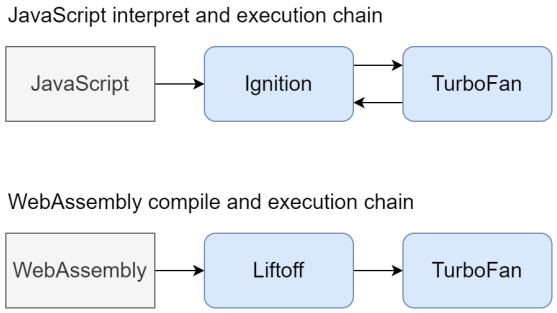

Figure 1: A simple overview of the different execution chains

First, consider the compilation and execution pipeline for JS in V8, an open source Javascript and WebAssembly engine powering all chromium based browsers like Google Chrome and Edge as well as the server-side JS environment Node for example. Note that there are many other engines but one can assume there are similarities between all advanced engines.

Before JS can be executed, it has to be interpreted (see figure 2). This is done in *Ignition*. The code is then passed to a code generator which generates unoptimized machine code. This machine code serves as a *Baseline* for the code execution engine. However, as you can see in figure 2 this step is much more complicated for Javascript than it is for WASM. Furthermore, to ensure that the execution is as fast as possible, the code is then passed onto *TurboFan* which is responsible for optimizing the code. As the unoptimized code is constantly being re-optimized and replaced this results in the execution leaving the *fast-lane* of execution. *TurboFan* uses real time statistics of the JS execution to determine which parts of the code to focus on optimizing.

WebAssembly however, as previously mentioned is only compiled once into machine code and optimized in runtime. Thus meaning that code executed from WASM will never leave *TurboFan*, the *fast-lane*. Making it significantly faster than JS (Google (2020)).

However, it is not only because of it's speed WASM is interesting for developers. It is also enables programmers to be able to run other programming languages, at near native speed, in the web browser, along with the results of sophisticated optimization engines part of their respective compilers. This allows developers to do things that has not been possible before. On the official WASM website you can read about a lot of different use cases that are possible with the

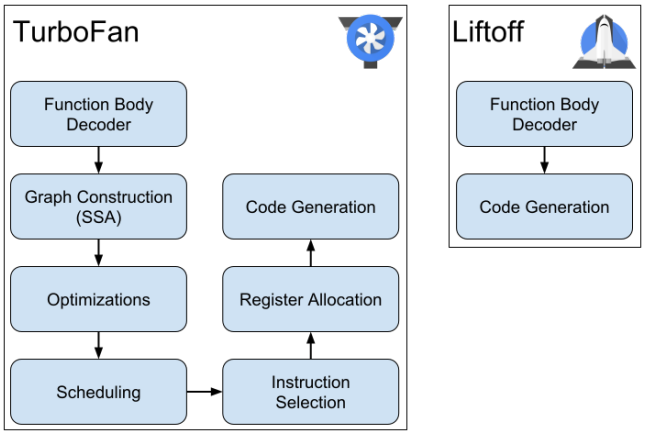

Figure 2: JS and WASM pipelines in the V8 engine.

help of the speed and compatibility provided with WASM. Image recognition, CAD applications, Virtual Machines and Encryption are a few examples that are made viable through the speed and efficiency provided from WASM.

Today however, WASM is not all sunshine and rainbows. As with all bleeding edge technologies, there are quirks and problems with the using it before it is a well established and used technology. First and foremost, although being around for more than 3 years, there is not nearly as much documentation for WASM as there is for other web-languages. This makes it difficult for the majority of developers to get started with the language. It is simply not worth the hassle, yet. As the language gets more and more used, it will gain a bigger user base and this problem will fade away.

Furthermore, it's not only the lack of documentation that prevents WASM to get a broad and big user base in today's situation. There are some limitations within the language that makes WASM harder to work with than the majority of web developers would like. As of writing this report there is no garbage collection in WASM. There is no exception handling either. Both of these are in development and we will likely see many changes coming to WebAssembly in the coming years.

Even if these language specific limitations does not scare the developer away, the fact that it is hard to debug just might. Unfortunately there are no easy ways to debug WASM applications. The best way to today is the good old print statements. It is not the most efficient way to debug applications but it is, to be fair, still widely used amongst developers today.

Problems and limitations aside, WASM is arguably the holy grail of web game development because of its speed and possibility to compile languages like Python, Rust or C# into WASM.

3.3 Sandboxing as a security mechanism

Executing code given to you from an external source that you do not have control over usually exposes you to a ton of potential security risks. One of the main risks is the program doing something you do not expect, like accessing a resource.

Controlling exactly what the program can and cannot do is very complicated. One way to try and get around this is to execute the program in a very limited environment, knowing as sandboxing. This commonly used in malware analysis for example. To understand how a malware works you obviously do not want to just run it on your computer, it will do malicious things. Instead, you often use a virtual machine simulating the target operating system and run it there. In the VM you have much more control over the permissions of the system. You can for example limit all network access for the VM, restore the state of the machine to some previous point in time, and much more. Another example is Bubblewrap which can be used to sandbox processes. It uses Linux namespaces so isolate the program from the host filesystem, network, and etc. Notably, this sandbox utility does not require root privileges to run.

A browser is also a type of sandbox. Generally, each site is completely isolated from all other tabs and cannot access anything on the host machine unless given explicit permission (like a file from a user upload). The Javascript is executed in this sandboxed environment, without any access to the underlying operating system or host machine. The browser, however, cannot protect JS from itself. If a server serves a JS file the browser has no choice but to trust it. While there are many security policies in place like SOP, the web is still full of vulnerabilities. Most, if not all, moderately complicated websites will rely also on several third-party JS libraries, all of which can be potentially insecure. When importing a library, or any third-party JS code, it is difficult to restrict what resources it has access to. Generally, it has access to everything the other JS has access to.

There have been several attempts at fixing this issue. One solution might be to just sandbox the Javascript within Javascript. Terrace et al. proposed such a solution, *js.js*, in 2012. Their implementation uses a JS interpreter implemented in Javascript to safely execute third party scripts. This completely isolates third-party code in a sandboxed environment. While effective according to the authors it has a few major drawbacks. According to their paper it imposes a major performance penalty. Upwards of a 500 times slow down compared to regular JS execution is reported which is unsustainable for real world use. Their solution does however demonstrate a very interesting and promising solution, and without the major performance drawback a very plausible one. If one could only achieve this sandboxing without the performance drawback third-party code could be much more safely executed. This is were WebAssembly can help.

3.4 WASM's sandboxing functionality

One important motivation behind WASM and one of it's most important features is it's security. WebAssembly modules only have access to the resources given to the environment by the user. This means all WASM code is, fundamentally by design, executed in a strict sandboxing environment. This makes for a very secure code executing environment, enabling you to confidently use third party modules and share your own. The mechanism behind this is how WASM can interact with the outside executing environment. The only way for a WASM module to interact with the "outside world" is through three means: exported functions, imported functions, and a shared memory buffer.

Firstly, exported functions are functions compiled to WASM that the module chooses to expose to the outside environment. These can be used to give

functionality to the user of WASM, like for example exposing a much faster encryption implementation for use in the browser than what JS can offer. The module could then expose a function "encrypt" that encrypts an array of bytes. As with any third-party generated function the output of these have to be used carefully.

Secondly, we have imported functions. This gives the module additional capabilities. It is through these functions you can give a WASM module additional capabilities, by for example wrapping file access in an imported functions. These are functions that you as the WASM executor has to supply to the module upon instantiation. These imported functions, along with their signatures, are statically defined in the WASM binary. For example, a frontend framework compiled to WASM might need to import functions that wrap DOM manipulation. Otherwise it would be impossible for the WASM module to change the DOM by itself. An executor of this WASM module would then have to explicitly supply these functions upon creation of the WASM module. A third-party JS library, in contrast, does not need to be supplied functions for DOM manipulation. It as full access to the DOM by default, as well as the ability to do HTTP requests and much more. Here is where the WASM sandboxing shines and shows how WASM can be a much more secure environment for third-party code than plain JS. If you do not supply an imported function to do HTTP requests you can be sure that the WASM module will not do any HTTP requests. As simple as that.

Lastly, we have the shared memory buffer. This is a potential source of vulnerabilities. When reading from this buffer the executor of the module has to be very careful. It cannot implicitly trust the data there. A trivial example would be interpreting the buffer as an ASCII string and passing it directly to the "eval" function. This essentially gives the WASM module arbitrary code execution and would be a huge vulnerability. While obviously a contrived example, one could imagine less obvious situations were the executor of the module has to be careful with how it uses and interprets data produced by the WASM module.

4 Description of work

You can find the demo hosted at https://wasm-tictactoe.herokuapp.com and the full source code at https://github.com/AxlLind/wasm-sandbox-demo. Instructions for how to run the application locally can be found in the Github repository. On the surface the demo is very simple. It allows users to put bots against each other to play tic-tac-toe. On the landing page (see figure 3) the user is presented with two drop downs where they can choose two bots to play against each other, or choose "human" to play manually. After pressing the "Play The Game" button the user will get redirected to a page where the selected bots play tic-tac-toe versus each other (see figure 5).

Figure 3: The demo's landing page

Where the demo gets interesting is how bots are created (see figure 4). Any user is free to upload their own bot which, if we were using JS for this, would be an excellent way to facilitate XSS. Instead the user uploads their own bot written in any language they want and compiled to WebAssembly. Due to WASM's previously mentioned sandboxing functionality this is completely safe. In the backend we validate that the WASM module looks like we expect. We expect the file to require no imports and to only export a single function called "makeMove". This function should take a single $i32$ as input which contains the board representation and should output a number between 0-8, indicating the chosen move.

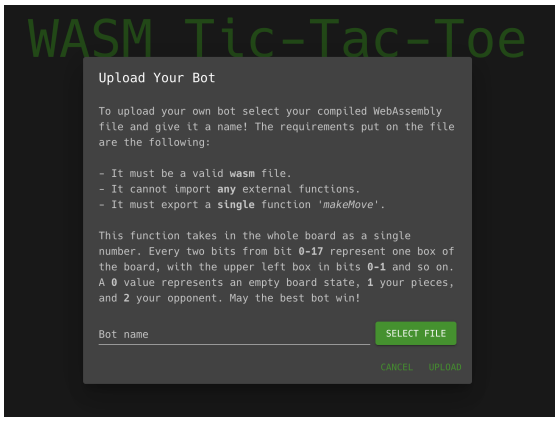

Figure 4: Modal presented to the user when uploading a new bot.

4.1 Frontend

The frontend is written in React utilizing modern React features like functional components and hooks. We have both done a lot of frontend work in React before. While maybe slightly overkill for this demo, it is a way to create websites we both feel very comfortable with. For easy of use, we used a library called Material UI. This is a React component library based on Google's Material

Design which makes creating good looking websites very easy. Most of the files are related to the UI of the website and are thus somewhat irrelevant for this course.

The code of most relevance for this project is the code handling the WebAssembly integration. This is all abstracted into a Bot class, the code of which can be found in the Bot.js file. Upon instantiation of a Bot, it simply downloads the WASM file from the backend based on the bot name. This is stored in base64 so it first has to be decoded and stored in an ArrayBuffer. We then compile the received bytes into a WASM module via the web standard api WebAssembly.instantiate() provided by the browser. Since we on the backend have verified that the module looks like we expect we can safely call the modules $makeMove$ function.

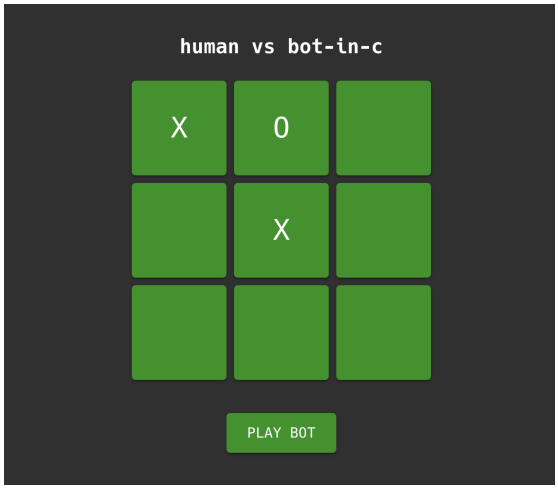

Figure 5: Bots playing tic-tac-toe versus each other.

4.2 Backend

The backend is a simple server that allows users to upload base64 encoded bots with a name. We chose to write the code using the framework FastAPI since it is easy and fast to develop application in and would suit our needs perfectly.

The backend also verifies the structure of the received WebAssembly file. This code can be found in validate_wasm.py. It verifies that the file is in fact valid WASM, that the module does not import any functions, that it only exports a single function, and that the exported function is called "makeMove". This ensures both that the module is usable for us in this application and that it satisfies our security constraints.

The backend is hosted at Heroku on the following url:
https://wasm-bots.herokuapp.com and the API documentation is found at:
https://wasm-bots.herokuapp.com/docs

4.3 Bots compiled to WASM

For our demo we developed three bots, written in three different languages. This was done for fun and as a learning opportunity for us, as well as a demonstration of WASM's ability to bring many other languages into the browser. While arguably outside the scope of this project, we also developed some more advanced bots using classic algorithms for zero-sum games.

- The first one, written in C, was a trivial example just to get up and running. It just selects the middle square if available and otherwise selects the first free square it finds. This was primary used during testing and development. See trivial-bot.c.
- The second bot, written in C++, uses a heuristic to give each possible board a score on how good it is for the bot. It then chooses the move which maximizes this score. This heuristic is based in giving each row, column, and diagonal a score based in the pieces involved. See heuristic.cpp.
- The third bot, written in Rust, uses a variant of the Minimax algorithm together with unlimited look ahead of moves to find the optimal move for all possible boards. This was also an interesting demonstration of the speed of WASM. It is able to do a complete exhaustive search from an empty board, which effectively checks every possible tic-tac-toe board, in around 50ms. See optimal.rs.

Because of the nature of tic-tac-toe, it is impossible to devise a guaranteed winning strategy. At most you can force a draw, and win against bad opponents. The last two bots, therefore, guarantees to never lose but cannot not guarantee winning.

To compile these bots to WASM, we used a web-based utility called WebAssembly Studio. This is a web application which can be used to compile small programs into WASM. It currently supports C, C++, Rust, Typescript, and raw WebAssembly in the text format. Using WebAssembly Studio eliminated the the need to set up complicated tool chains locally, figuring out correct compiler flags and so on for each language we wanted to compile. This enabled quick prototyping and for us to use more languages to make bots. To use this utility we just created a new project for each language and uploaded the code. We then just downloaded the compiled WASM file from the site.

5 Conclusions

Our demo clearly demonstrates the sandboxing security of WASM modules. Even though we execute user uploaded code in client of any other user the website is not vulnerable to XSS. Using JS, this would be a textbook trivial example of cross-site scripting. If the bots were implemented using Javascript you could easily execute HTTP requests, read the users cookies, and much more in the "makeMove" function. We do not allow any imports into the user-provided WASM module which makes it impossible for the module to perform any restricted actions, guaranteed by the WASM executor implementation (the browser in this case). It can only do computational tasks. The module is essentially executing in a zero-permission environment, ensured by the WASM

sandboxing feature. To show that it is not vulnerable to XSS let us consider the modules point of contact:

- *Imported functions.* As previously noted, the backend explicitly prohibits any module that imports any functions.
- *Exported functions.* The only exported function is "makeMove", also guaranteed by the backend's validation. It has a single int as a return value. The only way this result is used is to index an array representing the current board state. As this is a demo we currently have very limited error handling. Even if the bot tried to supply a very large integer as an output, due to the nature of indexing arrays in JS this would just return undefined which could be dealt with.
- *The shared memory buffer.* This memory buffer is never read from by the client.

This website is obviously quite simple. There are no user accounts, or session cookies to steal for example. One could, however, imagine several XSS vulnerabilities exposed in a more real world scenario which is prevented by using WASM instead of JS in this application. The only vulnerability exposed by the WASM code is client-side DOS. This could be achieved by simply entering an infinite loop in the "makeMove" function. Doing so would freeze the client but in modern browsers this hardly creates a real security issue. The browser will simply alert the user and ask them if they want to close the tab. While annoying, it has no serious security implications. For this reason though, we never execute the WASM code on the server side as that would lead to a real DOS vulnerability. With a more sophisticated implementation you could on the other hand give a time limit to the execution time of the WASM module and kill it if it takes more than a certain amount of time but that seemed out of scope for this demo.

We feel that this tic-tac-toe demonstration clearly shows the promise of WASM from the security perspective. If the Javascript ecosystem moved towards compiled WASM modules for third-party code we would not only see a potentially faster web but also one with much less security issues.

References

Google (2020), 'V8 official developer website'. Fetched 2020-05-21.
URL: *https://v8.dev*

OWASP Foundation (2020), 'Owasp top ten'. Fetched 2020-05-21.
URL: *https://owasp.org/www-project-top-ten/*

Terrace, J., Beard, S. R. and Katta, N. P. K. (2012), Javascript in javascript (js. js): Sandboxing third-party scripts, *in* 'Presented as part of the 3rd {USENIX} Conference on Web Application Development (WebApps 12)', pp. 95–100.

W3Techs (2020), 'Usage statistics of javascript as client-side programming language on websites'. Fetched 2020-05-21.
URL: *https://w3techs.com/technologies/details/cp-javascript*

WASM Group (2020), 'Webassembly official website'. Fetched 2020-05-21.
URL: *https://webassembly.org/*

WebAssembly Working Group (2020), 'Webassembly specification'. Fetched 2020-05-24.
URL: *https://webassembly.github.io/spec/core/*

A Collaboration

Most of the backend code was done by Albin since he has a lot of experience from work with python backend code, with some code like the WASM validation done by Axel. On the frontend, Axel started with the initial implementation, setting up the landing page, upload modal, etc and Albin initially created the tic-tac-toe game. During a pair-programming session both of us implemented the WebAssembly data flow and implemented it into the game, which was the most relevant part to this report. Lastly Axel did some of the final touch ups both to the code and design of the website and Albin handled setting up and deploying it to Heroku, a cloud hosting platform. The tic-tac-toe bots were mostly done by Axel.

The report was written in a couple of sessions over voice calls were both Albin and Axel participated equally. Every section in the report is written by both parties. Some parts are written in majority by Albin and some parts are written in majority by Axel. We decided on an overall structure but did not decide who was going to write what. Instead, we merely just started writing and added content were it was needed. The report is written in latex using Overleaf as a shared real-time latex editor.