Script

Felix hatte auf einer Folie den Satz: “Development should be possible from any platform, for any platform”. Und perfekt daran schließt WebAssembly an.

Definition

- WebAssembly ist ein **Offener Standard**, welcher ermöglicht, **Software in irgendeiner Sprache** zu schreiben und diese dann über den Browser, mit **near-native-speed** auszuführen.
- Die ursprüngliche **Spezifikation** enthält eine **Assemblyartige-Sprache** Namens **WebAssemblyText** und die **WASM Binary** an sich, zu welcher WebAssemblyText kompiliert werden kann.
- Man wird Code allerdings **nicht mit der mitgelieferten Sprache schreiben**, sondern die **WebAssembly Binary** als **Compilationtarget** für andere Sprachen verwenden.
- E.g. ein **Spiel**, welches mit **Unity** und **C#** erstellt wurde im Browser ausführen. Das wurde übrigens auch mit dem Spiel AngryBots beim ursprünglichen WebAssembly-Pitch gemacht.

WebAssembly Text

- Im folgenden ein kleines Beispiel:
  - Wir sehen hier eine simples WASM module mit einer **add** function, welche zwei Parameter nimmt und diese addiert
  - **Module**: ist dann einfach eine stateless WASM Binary mit einer gewissen Menge an Code.
  - WebAssembly Text verwendet **S-Expressions**, welche ein altes Format ist um Baumstrukturen darzustellen
  - **Function signature**
  - **Function body**:
    - Stack Machine - Push zwei Values auf den Stack und addiere mit der i32 add Instruktion
    - i32 add: Poppe die zwei Values und Pushe das Ergebnis
  - **Export block**: “add” ist der identifier, welcher von JavaScript verwendet wird und $add ist intern, um die zu exportierende Funktion festzustellen
- Unsere exportierte Funktion könnte jetzt so aus JavaScript aufgerufen werden. Die **WebAssembly.instantiateSteaming** Funktion kompiliert und instantiiert ein WebAssememby Module direkt von einer gestreamten Quelle. Das ist der Effizienteste Weg, ein WASM-Modul zu laden.

Demo - Javascript Glue Code

- **JavaScript Code**, welcher nötig ist, da WASM nicht auf **Web APIs** zugreifen kann. Daher muss WASM JavaScript callen, welche dann den

API call machen. In e.g. **Emscripten** implementiert der glue code Zugriff auf **SDL, OpenGL, OpenAL und teile von POSIX**.
- Dieser Code immer **bereitgestellt**
- In Go erhalten wir den Glue Code aus der **Go installation** (`wasm_exec.js`)
- Ganz wichtig hierbei ist, dass sich dieser mit den **Go Versionen ändert**. Wenn also das Program so nicht bei euch läuft, versucht mal die `wasm_exec.js` durch eure eigene auszutauschen.

Demo - importObject

- Das **importObject** enthält dabei die **Werte**, die in die neue **Instanz** einfließen sollen, also **Funktionen** oder `WebAssembly.Memory` Objekte

Additional Information

- **Spectre** attack in 2018 auf den **SharedArrayBuffer**. Der Support für diesen wurde dann entfernt und wird jetzt wieder Stück für Stück hinzugefügt.

Terminology

- Table ist einfach ein **typisierter array von Referenzen** zu e.g. **Funktionen** oder anderen Informationen, welche aus **Sicherheitsgründen** nicht einfach so im **Linear Memory** gespeichert werden sollten
- **Instance**: **Module inklusive Memory, Table und Imports**

WASI

- Realisiert wird das ganze wie bei CloudABI's capability-orientiert, weswegen WASI auch gut in das Sanboxing Modell von WASM passt. WASI hat keine Möglichkeit nach außen zu kommunizieren ohne von außen mitgegebene capabilities e.g. File descriptors.
- E.g. Statt open, openat, hier wird dann ein FileDescriptor benötigt.
- Es gibt eine system call wrapper layer, welche calls zur eigentlicheen WASI Implementation macht, welche diese calls dann auf die Umgebung mapped
- Ziel damit ist eben portabilität und sandboxing auf system interface level
- Geplant ist eine full-featured libc implementation

Security

- Es gelten also die **gleichen Sicherheitsbeschränkungen** wie auch schon für JavaScript Code
  - **Same-Origin Policy**
  - Auf Dateisystem oder Hardware kann auch nur mit **Permission** zugegriffen werden

- Um nicht mit e.g. C++ oder C **beliebigen Speicher** e.g. Passwörter oder andere Tokens auslesen zu können werden WASM-Moudlen sperate Speicherbereiche zugewiesen
  - **Untermenge** des JS-Heaps und wird durch einen **ArrayBuffer** realisiert.
  - **JS-Umgebung** weiß daher zu jedem Zeitpunkt die **Größe** und **Inhalt** des Buffers -> Unerlaubte Speicherzugriffe unterbinden, da Zugriffe nur innerhalb des Buffers erlaubt sind
  - **Overhead**, aber sonst zu unsicher
- **Execution** Stack wird außerhalb des WebAssembly-Speichers gespeichert und WASM hat nur Lesezugriff
- Bei Function calls wird mit vorher erwähnten **Tabellen** gearbeitet
  - Statt wie normalerweise **Zieladresse** im call-Befehl verwendet der call Befehl **zwei Parameter**. Einen **Index und eine Funktionssignatur**. Der Index zeigt in die Tabelle mit **Funktionspointern**, die auch **außerhalb des WebAssembly-Speichers** gespeichert wird, sodass das Überschreiben nicht möglich ist.
  - **Prüfung** ob Funktionssignatur mit der an Index X übereinstimmt
    - Nur wenn ja, wird die **Funktion an der Adresse aufgerufen**
    - Wenn nein, wird das Modul **sofort gestoppt**
- **Traps**, **JavaScript Exceptions** um abnormales Verhalten an die Laufzeitumgebung zu melden. Module wird sofort gestoppt
- Trotz dessen will ich euch nicht vorgaukeln, dass WASM die **perfekt sichere Lösung** ist. Es gab natürlich auch mal **Sicherheitslücken**. **Crypto Mining**, **Side Channel Attacks Spectre Angriffe** und ausbrüche aus der **Sandbox** sind schon vorgekommen.