Script

Felix hatte auf einer Folie den Satz: “Development should be possible from any platform, for any platform”. Und perfekt auf dieser Grundphilosophie baut Wasm auf

Definition

- AngryBots

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 nach dem Kompilieren 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 - importObject

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

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.

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 eigentlichen WASI Implementation macht. WASI mapped dann diese calls auf die eigentliche Umgebung.
- Ziel damit ist **portabilität** und **sandboxing** auf system interface level
- Geplant ist eine full-featured **libc** implementation