WebAssembly

Wasm On the Edge

Prepared for syd<video> meet-up

Kev Staunton-Lambert
pyrmontbrewery.com

Okay, **what** is Wasm...

Cross platform binary standard to run **pre-compiled code** from any **web browser*** on any **target platform***

(* major browsers on the platforms they support: pc/mac/linux)

Safari

Microsoft Edge

Firefox

Google Chrome

what else?

… run **pre-compiled** code **server-side** via **Node.js** too!

```
node --expose-wasm
```

and rather excitingly edge compute…

… **sandboxed** user code running on the **CDN edge**!
e.g. manipulate files on-the-fly (not on the origin)

why that's a good thing...

Compiled code means **optimised** code
(typical performance is more than 4x faster than interpreted JavaScript)

Common op-codes now shared with **hardware architecture**
(improved h/w acceleration options, op-codes limited to sandbox)

Tiny code, fast to download, small storage/RAM footprint

Uses standard **LLVM** toolchain, so **C / C++**, **Rust**, **Go**...
(works well with compiled code but less so Java which relies on VM to optimise)

why that's a good thing… edge compute

Fastly CDN lets us run WebAssembly on the edge

Apple Low-Latency is a great use case where:

- Playlist can be generated away from the origin
- File parts could be generated rather than uploaded - virtual files chunked on the fly

We can also do other things like modify TS headers on-the-fly

why that's a good thing… edge compute

(example) HLS uses MPEG TS files

TS files are well structured (188 byte packets)

PES PTS/DTS and PCR headers are easily offset on the CDN edge, leaving the origin file untouched

ISOBMFF boxes also easily modifiyable

(can be used to remove/smooth out player discontinuities)

how it works: wasm assembler language

Wasm is a binary specification, but includes also a human readable text assembly format

(example) module with a function `getKev` that adds the numbers 2 and 4

kev.wast

```
(module
   (func (export "getKev") (result i32)
      i32.const 2
      i32.const 4
      i32.add
```

how it works: assemble a binary (**W**eb**A**ssembly **Bi**nary **T**oolkit)

Assemble Wasm text to binary using "Wabbit" tools

```
wat2wasm kev.wast > kev.wasm
```

Creates a small 42 byte binary file called **kev.wasm**

```
0061 736d 0100 0000 0105 0160 0001 7f03   .asm........`....
0201 0007 0a01 0667 6574 4b65 7600 000a   ........getKev...
0901 0700 4102 4104 6a0b                  ....A.A.j.
```

how it works: what's inside that wasm binary

```
0061 736d 0100 0000 0105 0160 0001 7f03   .asm........`....
0201 0007 0a01 0667 6574 4b65 7600 000a   ........getKev...
0901 0700 4102 4104 6a0b                  ....A.A.j.
```

Always starts with wasm magic value **0061 736d** (= .asm)

how it works: what's inside that wasm binary

```
0061 736d 0100 0000 0105 0160 0001 7f03   .asm.......`....
0201 0007 0a01 0667 6574 4b65 7600 000a   ........getKev...
0901 0700 4102 4104 6a0b                  ....A.A.j.
```

Then wasm version id `0100 0000 (= wasm version 1)`

how it works: what's inside that wasm binary

```
0061 736d 0100 0000 0105 0160 0001 7f03   .asm........`....
0201 0007 0a01 0667 6574 4b65 7600 000a   ........getKev...
0901 0700 4102 4104 6a0b                  ....A.A.j.
```

Then modules: types, exports, such at the function name
Globals,
Tables,
memories,
and

how it works: what's inside that wasm binary

```
0061 736d 0100 0000 0105 0160 0001 7f03   .asm........`....
0201 0007 0a01 0667 6574 4b65 7600 000a   ........getKev...
0901 0700 4102 4104 6a0b                  ....A.A.j.
```

… **functions** the op*code instructions section* inside the
getKey exported function:

how it works: those op-codes **4102 4104 6a0b**

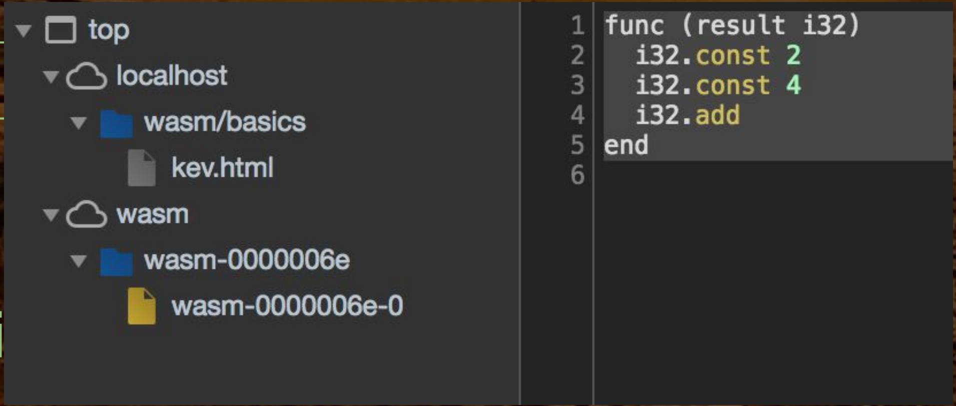
0x41 is op-code for **push** constant 32-bit integer onto the stack,
we call that twice… 02 + 04

```
0x41 02 -> i32.const [2]
0x41 04 -> i32.const [4]
```

0x6a is op-code for **add** from s

```
0x6a    -> i32.add [i32 i

0x0b    -> end
```



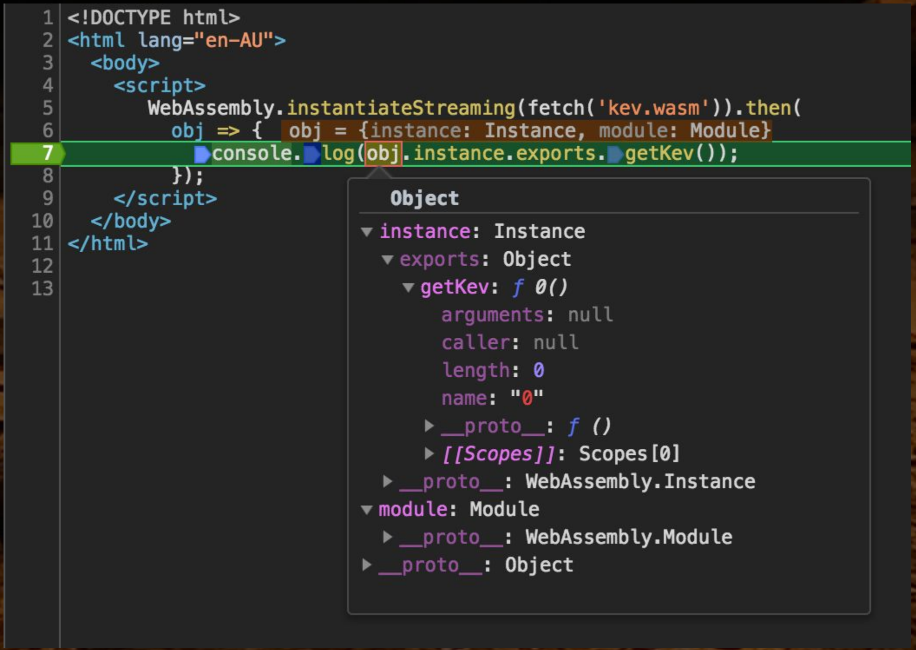
how it works: more sandboxed op-codes

Many other arithmetic operations and these usual suspects around control:

loop: a block with a label at the beginning which may be used to form loops
if: the beginning of an if construct with an implicit then block
else: marks the else block of an if
br: branch (aka goto) to a given label in an enclosing construct
br_if: conditionally branch to a given label in an enclosing construct

how it works: running wasm from JavaScript

```
<!DOCTYPE html>
<html>
 <body>
   <script>
WebAssembly.instantiateStreaming(
  fetch(‘kev.wasm’)).then(
  obj => {
    console.log(
      obj.instance.exports.getKev());
  });
   </script>
 </body>
</html>
```



Console log shows 6

how safe is it? All seems a bit dangerous…

It's not so bad, many implementations so expect some holes, but also quicker patching with community resolution

All memory/table marshalling well defined and type checked

Modules runs in a sandbox and will simply throw a JavaScript exception if something unexpected/naughty happens

Also utilises clang CFI (Control Flow Integrity) checking
https://clang.llvm.org/docs/ControlFlowIntegrity.html

what doesn’t work so well *yet*

Debugging symbols are currently missing
You *can* debug Wasm e.g. with WebtKit inspector, but this is not stepping the original sources (gdb) which can make things tricky, printf remains your friend

Shared objects (libraries)
You can’t link libraries, everything needs to be compiled into same global module right now

Support in SmartTVs, Roku, tvOS, Chromecast etc
Someday maybe :-) Embedded systems like these would

tools: emscripten: C/C++ -> Wasm

Emscripten modifies LLVM (clang) output into Wasm

`brew install emscripten`

Works by overriding LLVM in `~/.emscripten`

```
LLVM_ROOT =
'/usr/local/opt/emscripten/libexec/llvm/bin'
```

kev.c

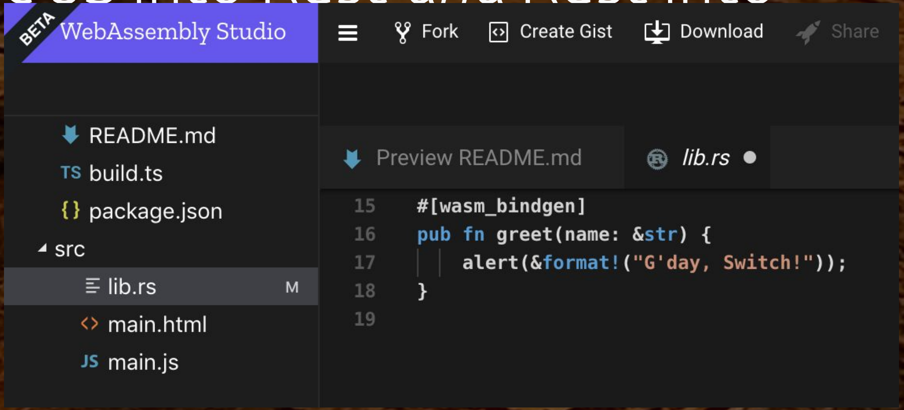
```
#include <stdio.h>
int main(int argc, char ** argv) {
  printf("G’day, Switch!\n");
}
```

tools: webassembly.studio: Rust -> Wasm

webassembly.studio is web based IDE tool
https://webassembly.studio/

It uses **wasm_bindgen** to generate the language bindings (which works both ways, expose JS into Rust *and* Rust into JS)

kev.rs
```
#[wasm_bindgen]
pub fn greet(name: &str) {
  alert(&format!("G'day, Switch!"));
}
```



tools: wasm-bindgen (JS to Rust bridge)

Installs using Rust's Cargo/crates.io package manager

```
curl https://sh.rustup.rs -sSf | sh

source $HOME/.cargo/env

rustup -v install  nightly
rustup target add wasm32-unknown-unknown --toolchain nightly
cargo +nightly install wasm-bindgen-cli
```

Make new project

```
cargo +nightly new kev_rust --lib
```

Then create Cargo.toml to pull Wasm package

tools: wasm-bindgen (Rust to JS bridge)

kev_rust/Cargo.toml

```
[package]
name = "kev_rust"
version = "0.1.0"
authors = ["Kev <kevleyski@gmail.com>"]

[dependencies]
wasm-bindgen = "0.2"
```

Build Wasm

```
cargo +nightly build --target wasm32-unknown-unknown
wasm-bindgen target/wasm32-unknown-unknown/debug/kev_rust.wasm
```

tools: some other Wasm editors

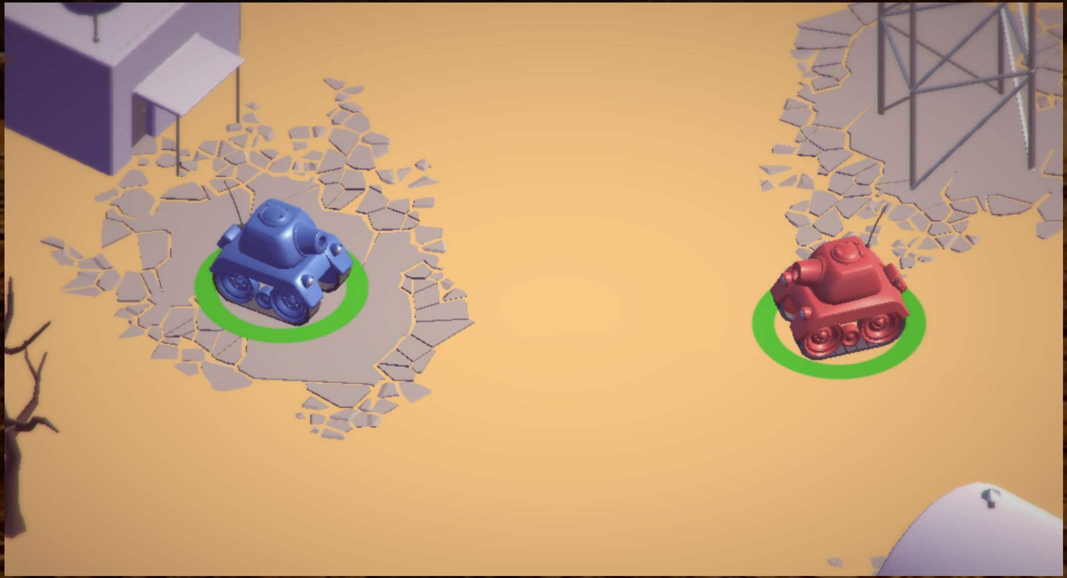
VSCode has a WebAssembly extension
https://webassembly.studio/

vi
https://github.com/rhysd/vim-wasm

WebKit inspector does pretty good job too (wasm/wast conversion etc)

wasm example: live interactive gaming

Unity3D WebGL
https://webassembly.org/demo/Tanks/



wasm example: CDN Edge compute

Modifying MPEG transport files on the edge
Manifest manipulation (ad insertion grep)

future ideas (disclaimer: Kev’s thoughts)

Skipping validation step by signing the Wasm binary?

There is a setup cost to running Wasm where it needs to check things are good each time it loads a binary, what if we can confirm we did that already with some sort of **code sign**? Code sign or registry check etc of course needs to be faster than validation itself without becoming an attack vector ;-)

Pre-known metadata could lead to less or **zero JavaScript**

Thanks!

Slides are here:

https://goo.gl/**2ahsEY**

Apple Low-Latency
https://tinyurl.com/**yyr2rz8m**
AV1 https://goo.gl/**pGnNgJ**

wasm@pyrmontbrewery.com.au

© 2007 Pyrmont Brewery