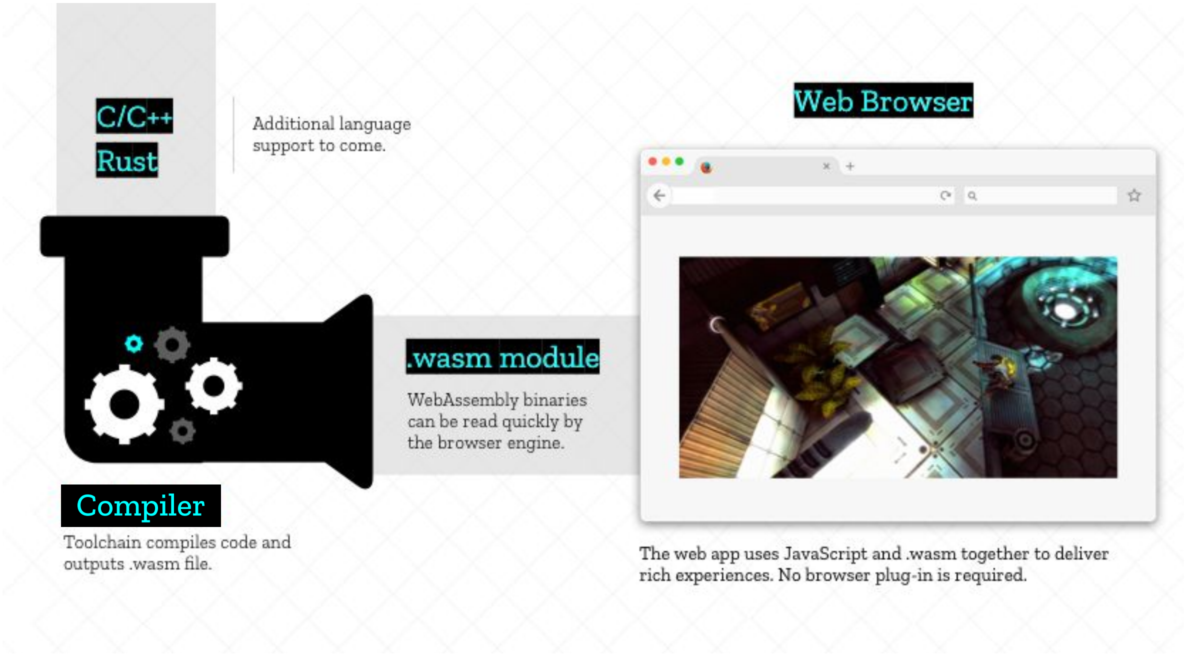
WebAssembly Outside the Browser

A New Foundation for Pervasive Computing

David Bryant
Fellow, Emerging Technologies
Mozilla
dbryant@mozilla.com

What Is WebAssembly (Wasm)?

- WebAssembly is a virtual CPU inside your browser
- Allows code in other programming languages to run in any browser
- Think of it as a virtual instruction set that runs efficiently on the web platform
- Why would we want to do that??


C/C++
Rust
Additional language support to come.
Web Browser
.wasm module
WebAssembly binaries can be read quickly by the browser engine.
Compiler
Toolchain compiles code and outputs .wasm file.
The web app uses JavaScript and .wasm together to deliver rich experiences. No browser plug-in is required.

A Brief History of WebAssembly

1995	2007	2013	2017	Today
JavaScript first released	HTML 5 begun	**asm.js** ships in Firefox 22	**WebAssembly** first ships in Firefox 52 in March Supported by all major browsers by November	**WebAssembly** supported by 91.39% of browsers globally 91.97% on desktop 92.95% on mobile

Web applications get steadily more powerful (and also steadily larger)

Web Games!

What’s So Great About WebAssembly?
(inside the browser)

- Fast, safe and portable
    - Executes with near-native performance and can leverage contemporary hardware
    - Code is validated and run in a sandboxed environment. No ambient access to the computing environment in which code is being run except through explicit permission.
    - Hardware, language, and platform independent
    - Open and standards based
- Efficient
    - Compact, so small to transmit (especially compared to text or native code)
    - Modular and parallelizable by design
    - Decoded, validated and compiled in a fast, single pass (and streamable to boot)
    - Intended to be easy to inspect and debug
- Supported by all the major browsers
- Can be targeted by a wide variety of programming languages and popular toolchains

Why Not Outside The Browser Too, Then?

- That’s a pretty good idea! Why not take advantage of a fast, scalable, secure way to run the same code across all machines?
- But Wasm gets a lot of benefit from running inside the browser, e.g.:
    - Access to and control of system resources (files, devices, operating system services, etc.)
    - Access to the network
    - Abstraction of the operating environment
    - Opaque abstractions enabled by the Wasm toolchain (e.g., Emscripten)
    - Execution supervision (threading and parallelism, scheduling, etc.)
- And software running outside the browser is different from software inside the browser, especially today
    - Comprised of a variety of programming languages
    - Comprised of shared components from a wide range of sources. This is both a need (reuse and ease of development) and a challenge (untrusted shared code and dependencies)

Thinking About The Problem
(in software today, generally)

- **Eighty percent** of code built into applications and services today comes from package registries like npm, PyPI and crates.io.
- Current software architectures don't make that kind of wide-ranging, open ended dependency safe, and that lack of safety puts our users at risk.
- We get some help from operating systems, and containers, but not enough
- Do you feel safe? Do you check all your dependencies, always?
- The very things that make Wasm work well inside the browser -- it's fast, safe, portable and efficient -- **could** make it a compelling way of running untrusted code anywhere, at scale.

Enabling Wasm Outside the Browser

- First problem -- accessing system resources, portably
- Operating Systems do this through a system call abstraction (POSIX)
- But POSIX depends on recompiling per platform and Wasm needs more. We need to go beyond POSIX and provide portable binaries
- Solution: **WASI**, the WebAssembly System Interface (announced in March 2019)
  - Compile once and run on many different machines
  - Capability-based security model
  - Avoids all kinds of distribution and deployment problems
  - https://wasi.dev/

Enabling Wasm Outside the Browser

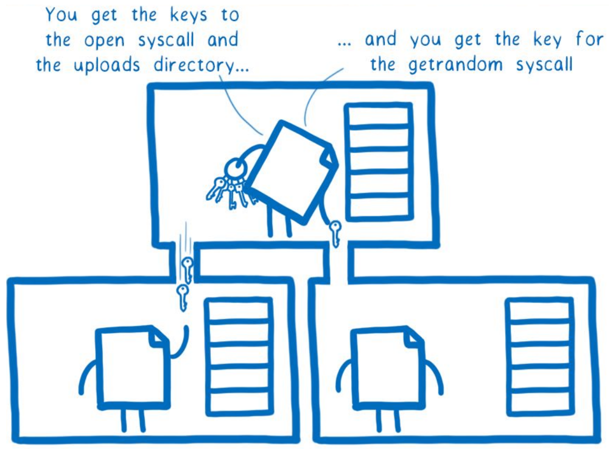
- Next challenge -- running WebAssembly modules from other languages
- Why would we want to do that?
  - Speed
  - Security
  - Portability
- Solution: **WebAssembly Interface Types** -- run Wasm from other languages like Python, Ruby and Rust (announced August 2019)
  - Wasm can only understand numbers (but has lots of numeric types)
  - Includes the means for efficiently mapping across different data types (via intermediate representation)

Enabling Wasm Outside the Browser

- And finally -- how do I actually run WebAssembly outside the browser?
- Solution: **Wasmtime** is our standalone, lightweight Wasm runtime. (There are others, too.)
- Wasmtime is built on our optimizing Cranelift code generator
    - Which is written in Rust
    - And which we've also incorporated in Firefox
    - Wasmtime supports WASI and WebAssembly Interface Types
    - And all the code is available on Github (of course)
- You can download Wasmtime and learn more from https://wasmtime.dev/
    - There's also a demo repository with examples of content to run, and ways to use Wasmtime
    - We're also maintaining WASI toolchains for Rust and C/C++

The WebAssembly “Nanoprocess”

- Start with a lightweight process model for basic isolation
- The add the advantages of WebAssembly:
  - **Sandboxing** -- a carry-over from inside the browser, with capability-based security
  - **Module-level memory encapsulation** defined into the standard (with no shared memory)
  - **Interface types** (highlighted earlier) for sharing of data without sharing memory and across different programming languages
  - A **system interface** designed around module-level permissions
  - A fine-grained way to cascade **capability-based permissions** to dependencies so the trust placed in them is precise and explicit
- Taken all together this approach to isolation results in the WebAssembly “**nanoprocess**”


You get the keys to the open syscall and the uploads directory...
... and you get the key for the getrandom syscall

Introducing the Bytecode Alliance

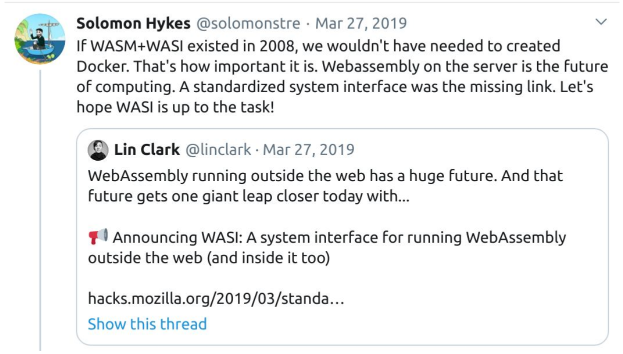
- There’s a lot of work to be done in making WebAssembly outside the browser a reality, though quite a bit of it was already underway here and there, e.g. Mozilla’s work on Cranelift
- Last November four companies already active with Wasm and WASI announced The Bytecode Alliance, their joint commitment to delivering a state-of-the-art runtime and language toolchains.
- Those founding members-- Mozilla, Fastly, Intel and Red Hat -- contributed open source project efforts to the Bytecode Alliance, including:
  - Wasmtime, from Mozilla
  - Lucet, Fastly’s ahead-of-time compiler and runtime focused on the kind of applications needed at the edge of the network.
  - WebAssembly Micro Runtime (WAMR), an interpreter-based Wasm runtime for embedded devices
  - Cranelift, a cross-platform code generator for Wasm, written in Rust
- More info at https://bytecodealliance.org/

fastly intel Red Hat moz://a

BYTECODE ALLIANCE

Interest and Adoption

- WebAssembly inside the browser is widely used for a variety of applications and services, ranging from games to complex applications to software frameworks (like Qt) and even to add low-level capabilities to browsers that otherwise wouldn't support them.
- Wasm outside the browser is attracting attention too
  - Solomon Hykes' tweet
  - Microsoft's Deis Labs has released 'Krustlet', which enables use of Wasm modules in Kubernetes to enable services smaller and faster than typical containers, with secure-by-default safety as well
- It's easy to get started
  - Great guides and API references on MDN
  - WebAssembly Guide for C/C++ developers
  - Wasm By Example on-line tutorial



Solomon Hykes @solomonstre · Mar 27, 2019

If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. Webassembly on the server is the future of computing. A standardized system interface was the missing link. Let's hope WASI is up to the task!

Lin Clark @linclark · Mar 27, 2019

WebAssembly running outside the web has a huge future. And that future gets one giant leap closer today with...

📢 Announcing WASI: A system interface for running WebAssembly outside the web (and inside it too)

hacks.mozilla.org/2019/03/standa…

Show this thread

Recap: Why We’re So Excited!

- WebAssembly has already proven its power and capabilities inside the browser, with more good examples coming every day.
- WebAssembly outside the browser can be a capable, secure platform that allows applications developers and service providers to confidently run untrusted code, on any infrastructure, for any operating system or device, leveraging decades of experience doing so inside web browsers.
- All the key elements are open source and standards based

“WebAssembly is changing the web, but we believe WebAssembly can play an even bigger role in the software ecosystem as it continues to expand beyond browsers. This is a unique moment in time at the dawn of a new technology, where we have the opportunity to fix what’s broken and build new, secure-by-default foundations for native development that are portable and scalable.

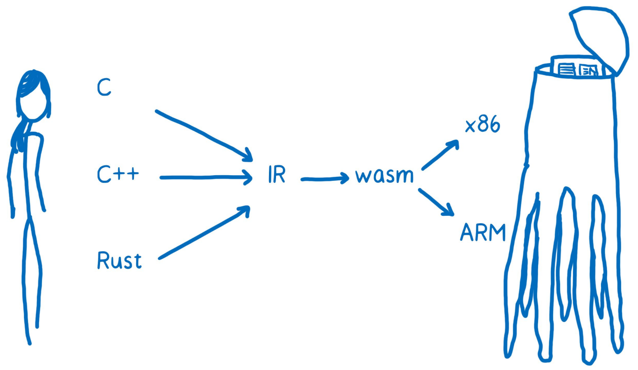
— Luke Wagner, Distinguished Engineer at Mozilla, co-creator of WebAssembly

Additional Resources

- Lin Clark’s marvelous “Code Cartoons” on WebAssembly
  - “A cartoon intro to WebAssembly” - a six part series that starts with a general overview, provides lots of context based on JavaScript, details working with Wasm modules, and talks about why it’s so fast. All published on Mozilla’s Hacks Blog for developers.
  - “Announcing the Bytecode Alliance”, illustrating the underlying vision and motivation
  - “Standardizing WASI”, where WASI was initially announced (and explained)
  - “WebAssembly Interface Types”, explaining the technical proposal

- Articles shared via the Bytecode Alliance’s website

- Many other posts on WebAssembly on Mozilla’s Hacks Blog



Q&A

(Thank you!)

dbryant@mozilla.com