HACKS

Search Mozilla Hacks

Standardizing WASI: A system interface to run WebAssembly outside the web

By Lin Clark

Posted on March 27, 2019 in Code Cartoons, Featured Article, and WebAssembly

Share This

Today, we announce the start of a new standardization effort—WASI, the WebAssembly system interface.

Why: Developers are starting to push WebAssembly beyond the browser, because it provides a fast, scalable, secure way to run the same code across all machines.

But we don't yet have a solid foundation to build upon. Code outside of a browser needs a way to talk to the system—a system interface. And the WebAssembly platform doesn't have that yet.

What: WebAssembly is an assembly language for a conceptual machine, not a physical one. This is why it can be run across a variety of different machine architectures.

Just as WebAssembly is an assembly language for a conceptual machine, WebAssembly needs a system interface for a conceptual operating system, not any single operating system. This way, it can be run across all different OSs.

This is what WASI is—a system interface for the WebAssembly platform.

We aim to create a system interface that will be a true companion to WebAssembly and last the test of time. This means upholding the key principles of WebAssembly—portability and security.

Who: We are chartering a WebAssembly subgroup to focus on standardizing WASI. We've already gathered interested partners, and are looking for more to join.

Here are some of the reasons that we, our partners, and our supporters think this is important:

Sean White, Chief R&D Officer of Mozilla

"WebAssembly is already transforming the way the web brings new kinds of compelling content to people and empowers developers and creators to do their best work on the web. Up to now that's been through browsers, but with WASI we can deliver the benefits of WebAssembly and the web to more users, more places, on more devices, and as part of more experiences."

Tyler McMullen, CTO of Fastly

"We are taking WebAssembly beyond the browser, as a platform for fast, safe execution of code in our edge cloud. Despite the differences in environment between our edge and browsers, WASI means WebAssembly developers won't have to port their code to each different platform."

Myles Borins, Node Technical Steering committee director

"WebAssembly could solve one of the biggest problems in Node—how to get close-to-native speeds and reuse code written in other languages like C and C++ like you can with native modules, while still remaining portable and secure. Standardizing this system interface is the first step towards making that happen."

Laurie Voss, co-founder of npm

"npm is tremendously excited by the potential WebAssembly holds to expand the capabilities of the npm ecosystem while hugely simplifying the process of getting native code to run in server-side JavaScript applications. We look forward to the results of this process."

So that's the big news! 🎉

There are currently 3 implementations of WASI:

- wasmtime, Mozilla's WebAssembly runtime
- Lucet, Fastly's WebAssembly runtime
- a browser polyfill

You can see WASI in action in this video:

And if you want to learn more about our proposal for how this system interface should work, keep reading.

What's a system interface?

Many people talk about languages like C giving you direct access to system resources. But that's not *quite* true.

These languages don't have direct access to do things like open or create files on most systems. Why not?

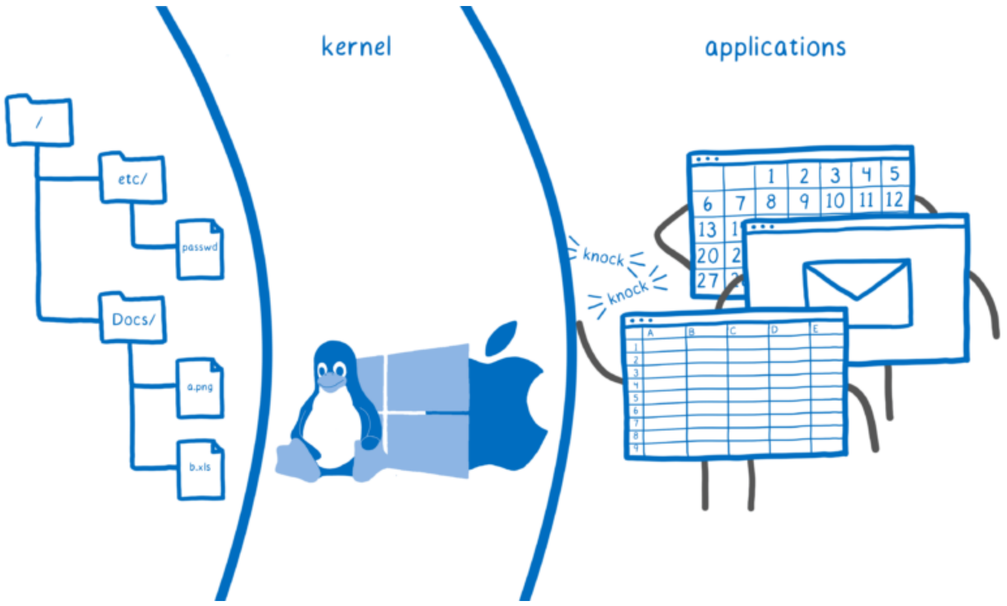
Because these system resources—such as files, memory, and network connections— are too important for stability and security.

If one program unintentionally messes up the resources of another, then it could crash the program. Even worse, if a program (or user) intentionally messes with the resources of another, it could steal sensitive data.

So we need a way to control which programs and users can access which resources. People figured this out pretty early on, and came up with a way to provide this control: protection ring security.

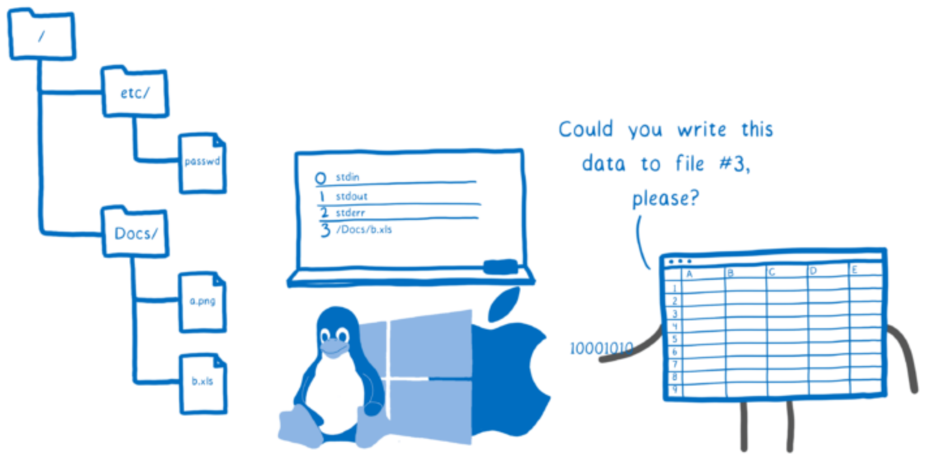
With protection ring security, the operating system basically puts a protective barrier around the system's resources. This is the kernel. The kernel is the only thing that gets to do operations like creating a new file or opening a file or opening a network connection.

The user's programs run outside of this kernel in something called user mode. If a program wants to do anything like open a file, it has to ask the kernel to open the file for it.



This is where the concept of the system call comes in. When a program needs to ask the kernel to do one of these things, it asks using a system call. This gives the kernel a chance to figure out which user is asking. Then it can see if that user has access to the file before opening it.

On most devices, this is the only way that your code can access the system's resources—through system calls.



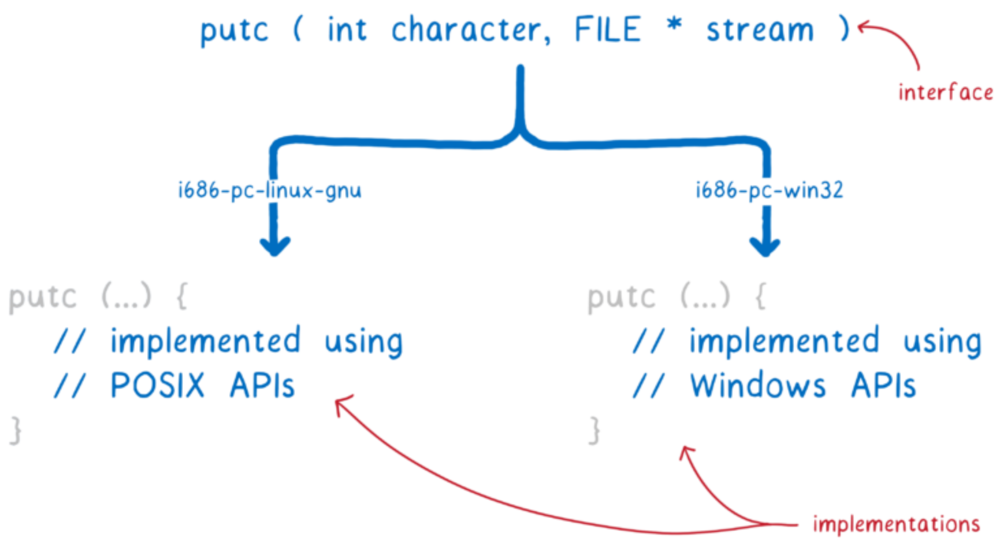
The operating system makes the system calls available. But if each operating system has its own system calls, wouldn't you need a different version of the code for each operating system? Fortunately, you don't.

How is this problem solved? Abstraction.

Most languages provide a standard library. While coding, the programmer doesn't need to know what system they are targeting. They just use the interface.

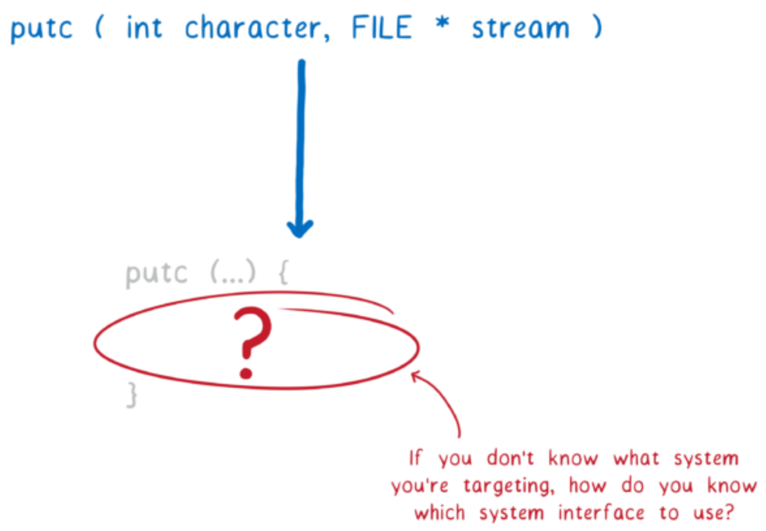
Then, when compiling, your toolchain picks which implementation of the interface to use based on what system you're targeting. This implementation uses functions from the operating system's API, so it's specific to the system.

This is where the system interface comes in. For example, `printf` being compiled for a Windows machine could use the Windows API to interact with the machine. If it's being compiled for Mac or Linux, it will use POSIX instead.



This poses a problem for WebAssembly, though.

With WebAssembly, you don't know what kind of operating system you're targeting even when you're compiling. So you can't use any single OS's system interface inside the WebAssembly implementation of the standard library.


putc (int character, FILE * stream)
putc (...) {
?
}
If you don't know what system you're targeting, how do you know which system interface to use?

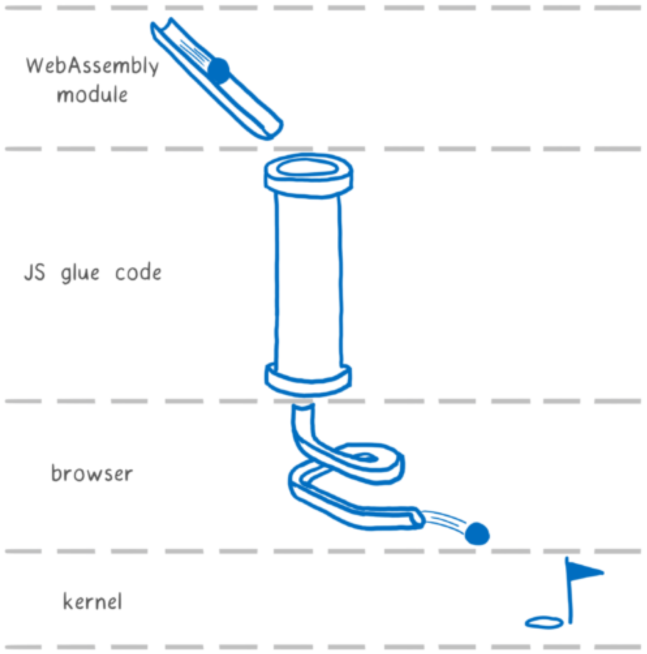
I've talked before about how WebAssembly is an assembly language for a conceptual machine, not a real machine. In the same way, WebAssembly needs a system interface for a conceptual operating system, not a real operating system.

But there are already runtimes that can run WebAssembly outside the browser, even without having this system interface in place. How do they do it? Let's take a look.

How is WebAssembly running outside the browser today?

The first tool for producing WebAssembly was Emscripten. It emulates a particular OS system interface, POSIX, on the web. This means that the programmer can use functions from the C standard library (libc).

To do this, Emscripten created its own implementation of libc. This implementation was split in two—part was compiled into the WebAssembly module, and the other part was implemented in JS glue code. This JS glue would then call into the browser, which would then talk to the OS.

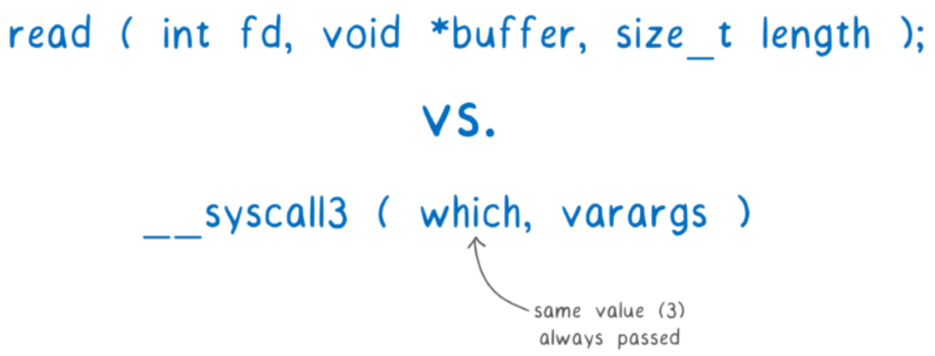


Most of the early WebAssembly code was compiled with Emscripten. So when people started wanting to run WebAssembly without a browser, they started by making Emscripten-compiled code run.

So these runtimes needed to create their own implementations for all of these functions that were in the JS glue code.

There's a problem here, though. The interface provided by this JS glue code wasn't designed to be a standard, or even a public facing interface. That wasn't the problem it was solving.

For example, for a function that would be called something like `read` in an API that was designed to be a public interface, the JS glue code instead uses `_system3(which, varargs)`.

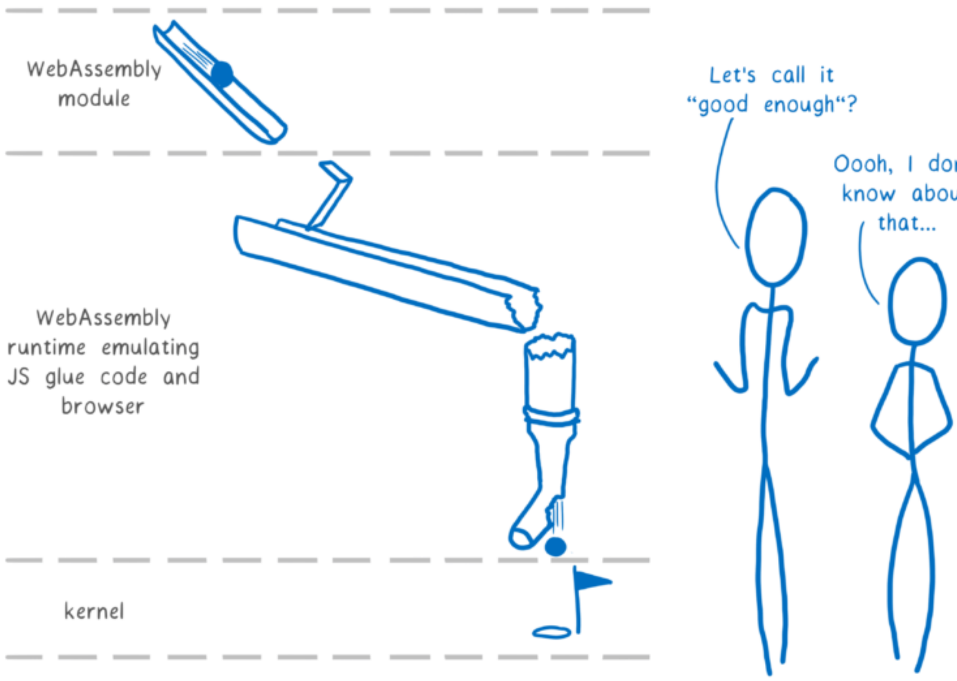

read (int fd, void *buffer, size_t length);
vs.
__syscall3 (which, varargs)
same value (3) always passed

The first parameter, `which`, is an integer which is always the same as the number in the name (so 3 in this case).

The second parameter, `varargs`, are the arguments to use. It's called `varargs` because you can have a variable number of them. But WebAssembly doesn't provide a way to pass in a variable number of arguments to a function. So instead, the arguments are passed in via linear memory. This isn't type safe, and it's also slower than it would be if the arguments could be passed in using registers.

That was fine for Emscripten running in the browser. But now runtimes are treating this as a de facto standard, implementing their own versions of the JS glue code. They are emulating an internal detail of an emulation layer of POSIX.

This means they are re-implementing choices (like passing arguments in as heap values) that made sense based on Emscripten's constraints, even though these constraints don't apply in their environments.


WebAssembly module
WebAssembly runtime emulating JS glue code and browser
kernel
Let's call it "good enough"?
Oooh, I don't know about that...

If we're going to build a WebAssembly ecosystem that lasts for decades, we need solid foundations. This means our de facto standard can't be an emulation of an emulation.

But what principles should we apply?

What principles does a WebAssembly system interface need to uphold?

There are two important principles that are baked into WebAssembly :

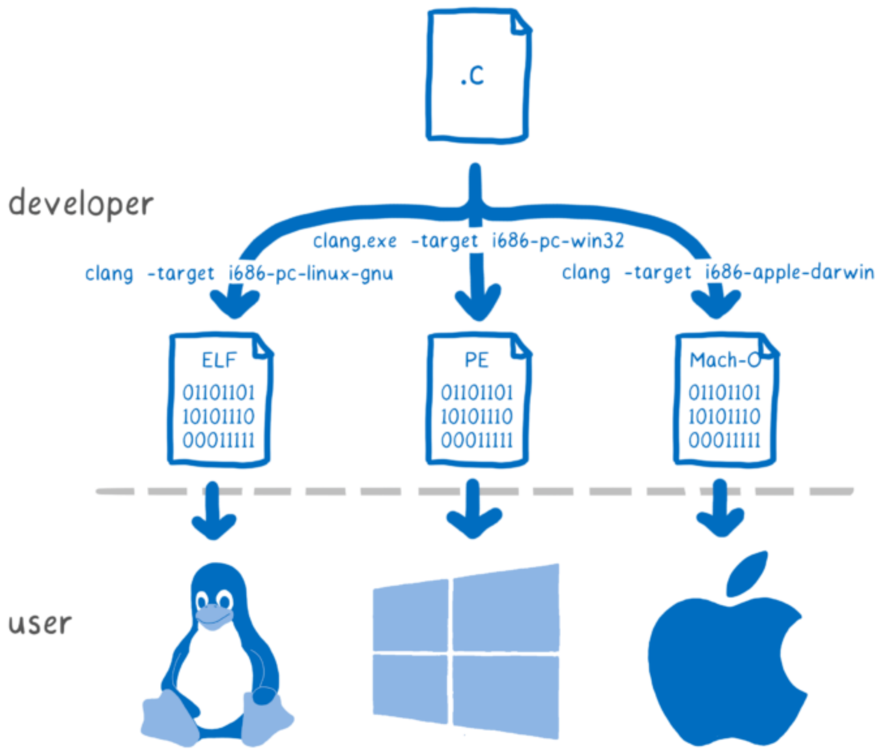
- portability
- security

We need to maintain these key principles as we move to outside-the-browser use cases.

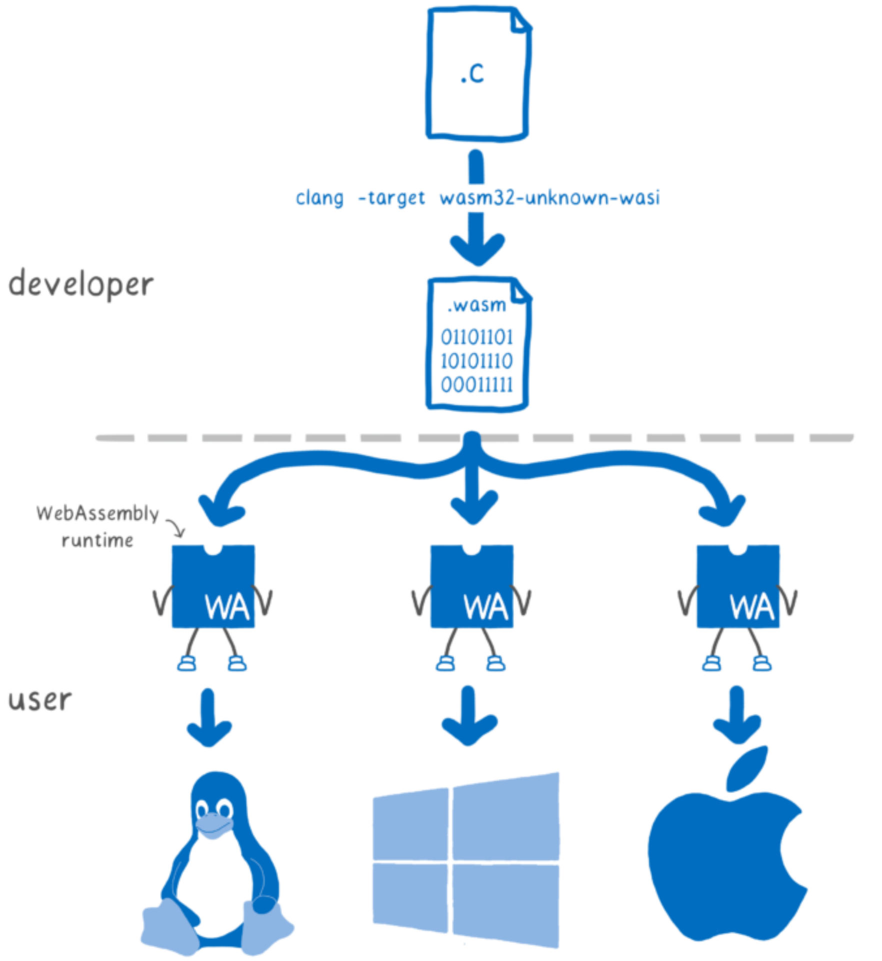
As it is, POSIX and Unix's Access Control approach to security don't quite get us there. Let's look at where they fall short.

Portability

POSIX provides source code portability. You can compile the same source code with different versions of libc to target different machines.



But WebAssembly needs to go one step beyond this. We need to be able to compile once and run across a whole bunch of different machines. We need portable binaries.



This kind of portability makes it much easier to distribute code to users.

For example, if Node's native modules were written in WebAssembly, then users wouldn't need to run node-gyp when they install apps with native modules, and developers wouldn't need to configure and distribute dozens of binaries.

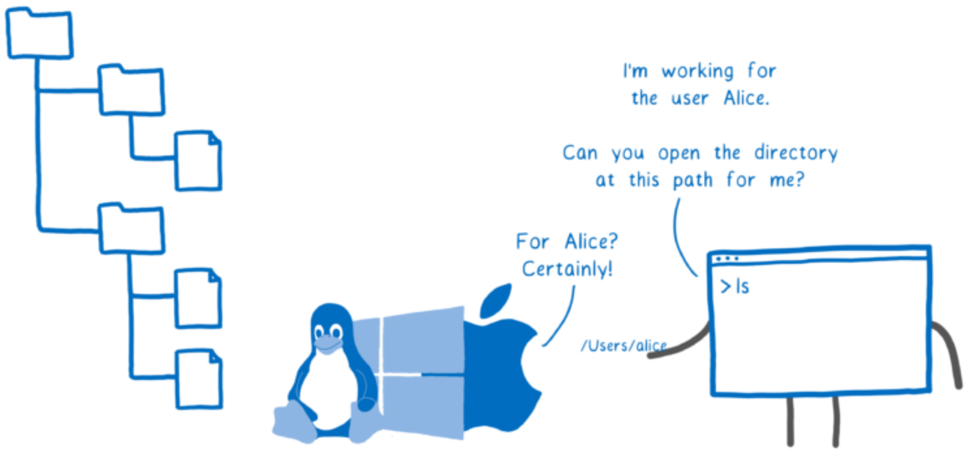
Security

When a line of code asks the operating system to do some input or output, the OS needs to determine if it is safe to do what the code asks.

Operating systems typically handle this with access control that is based on ownership and groups.

For example, the program might ask the OS to open a file. A user has a certain set of files that they have access to.

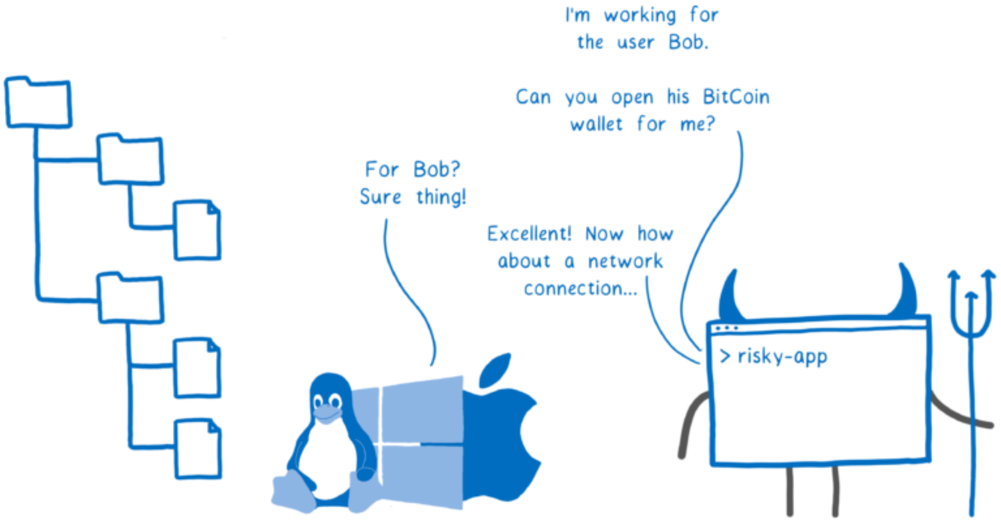
When the user starts the program, the program runs on behalf of that user. If the user has access to the file —either because they are the owner or because they are in a group with access—then the program has that same access, too.


I'm working for the user Alice.
Can you open the directory at this path for me?
For Alice? Certainly!
/Users/alice
>ls

This protects users from each other. That made a lot of sense when early operating systems were developed. Systems were often multi-user, and administrators controlled what software was installed. So the most prominent threat was other users taking a peek at your files.

That has changed. Systems now are usually single user, but they are running code that pulls in lots of other, third party code of unknown trustworthiness. Now the biggest threat is that the code that you yourself are running will turn against you.

For example, let's say that the library you're using in an application gets a new maintainer (as often happens in open source). That maintainer might have your interest at heart… or they might be one of the bad guys. And if they have access to do anything on your system—for example, open any of your files and send them over the network—then their code can do a lot of damage.


I'm working for the user Bob.
Can you open his BitCoin wallet for me?
For Bob? Sure thing!
Excellent! Now how about a network connection...
> risky-app

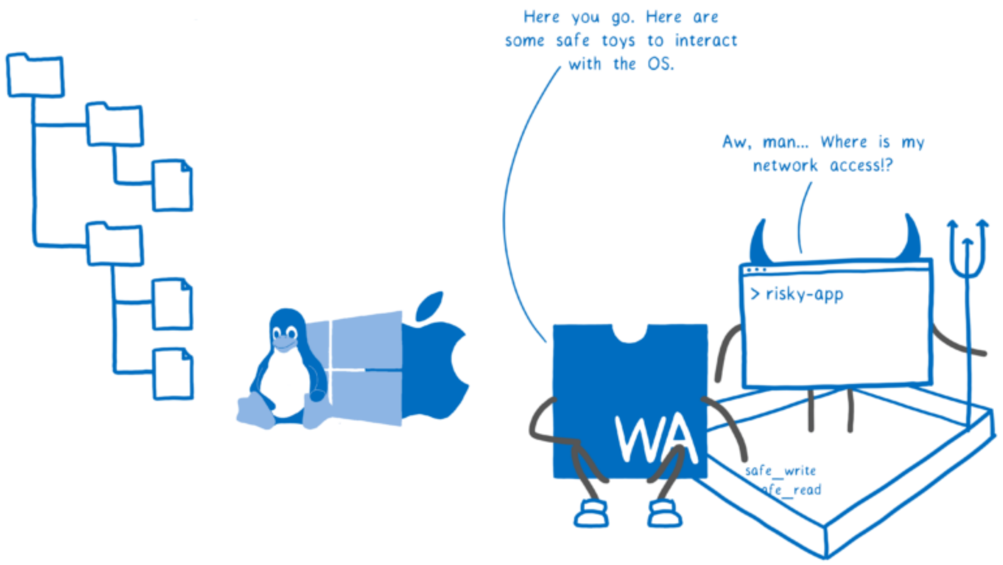
This is why using third-party libraries that can talk directly to the system can be dangerous.

WebAssembly's way of doing security is different. WebAssembly is sandboxed.

This means that code can't talk directly to the OS. But then how does it do anything with system resources? The host (which might be a browser, or might be a wasm runtime) puts functions in the sandbox that the code can use.

This means that the host can limit what a program can do on a program-by-program basis. It doesn't just let the program act on behalf of the user, calling any system call with the user's full permissions.

Just having a mechanism for sandboxing doesn't make a system secure in and of itself—the host can still put all of the capabilities into the sandbox, in which case we're no better off—but it at least gives hosts the option of creating a more secure system.


Here you go. Here are some safe toys to interact with the OS.
Aw, man... Where is my network access!?
> risky-app
WA
safe_write
safe_read

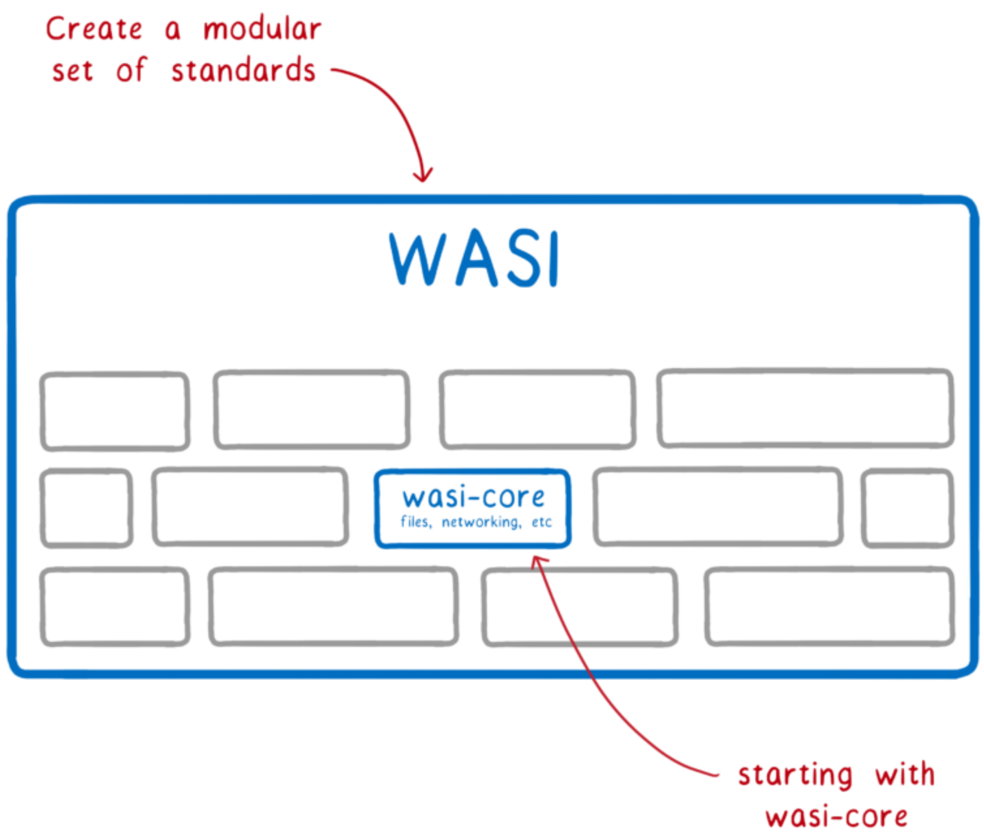
In any system interface we design, we need to uphold these two principles. Portability makes it easier to develop and distribute software, and providing the tools for hosts to secure themselves or their users is an absolute must.,

What should this system interface look like?

Given those two key principles, what should the design of the WebAssembly system interface be?

That's what we'll figure out through the standardization process. We do have a proposal to start with, though:

- Create a modular set of standard interfaces
- Start with standardizing the most fundamental module, wasi-core



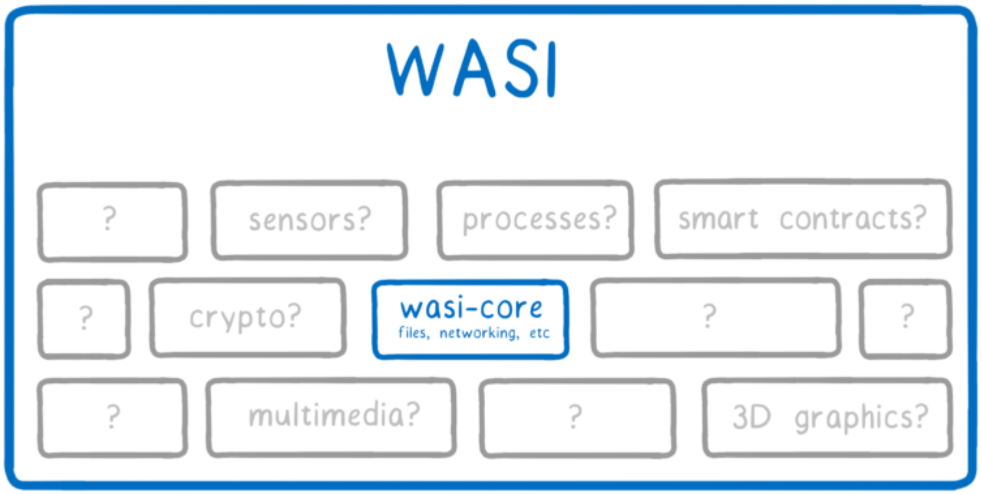
What will be in wasi-core?

wasi-core will contain the basics that all programs need. It will cover much of the same ground as POSIX, including things such as files, network connections, clocks, and random numbers.

And it will take a very similar approach to POSIX for many of these things. For example, it will use POSIX's file-oriented approach, where you have system calls such as open, close, read, and write and everything else basically provides augmentations on top.

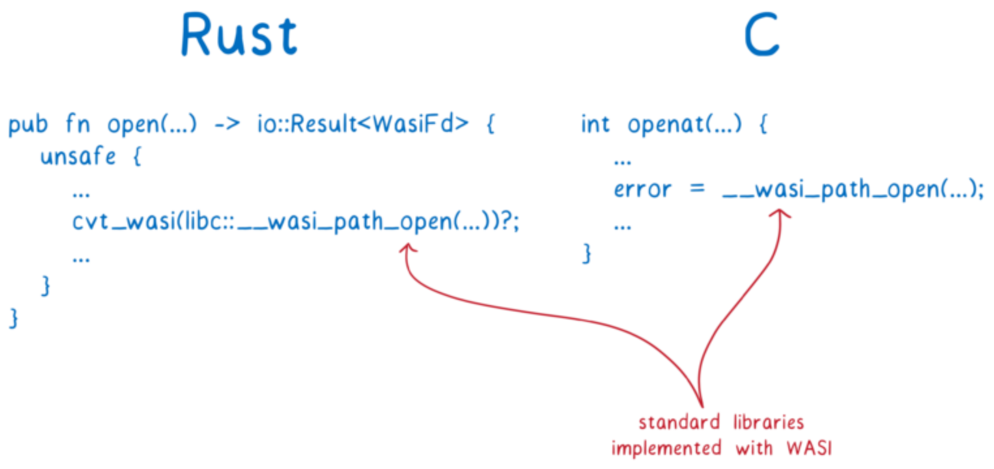
But wasi-core won't cover everything that POSIX does. For example, the process concept does not map clearly onto WebAssembly. And beyond that, it doesn't make sense to say that every WebAssembly engine needs to support process operations like `fork`. But we also want to make it possible to standardize `fork`.

This is where the modular approach comes in. This way, we can get good standardization coverage while still allowing niche platforms to use only the parts of WASI that make sense for them.



Languages like Rust will use wasi-core directly in their standard libraries. For example, Rust's `open` is implemented by calling `__wasi_path_open` when it's compiled to WebAssembly.

For C and C++, we've created a wasi-sysroot that implements libc in terms of wasi-core functions.



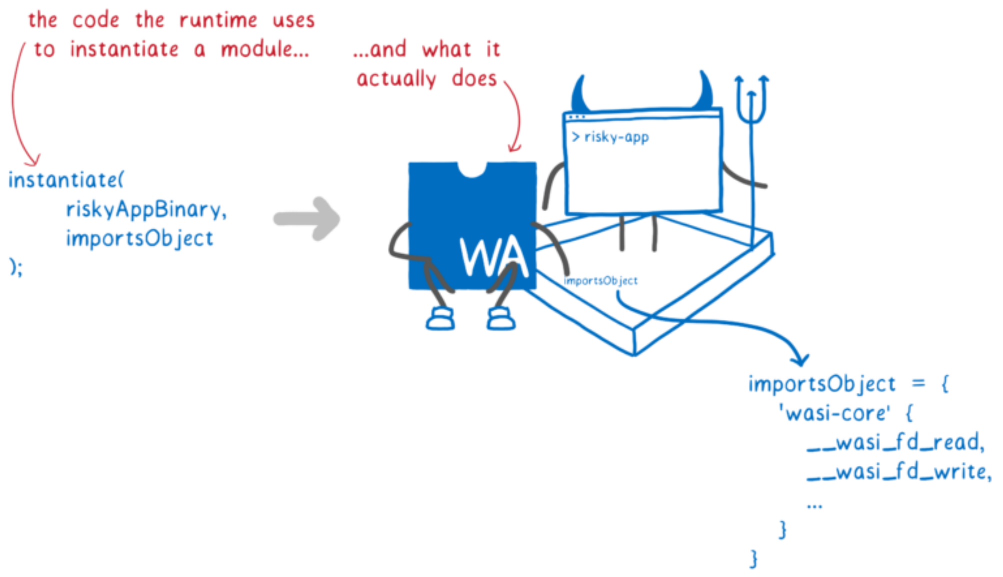
```
pub fn open(...) -> io::Result<WasiFd> {
   unsafe {
      ...
      cvt_wasi(libc::__wasi_path_open(...))?;
      ...
   }
}
```

```
int openat(...) {
   ...
   error = __wasi_path_open(...);
   ...
}
```

We expect compilers like Clang to be ready to interface with the WASI API, and complete toolchains like the Rust compiler and Emscripten to use WASI as part of their system implementations

How does the user's code call these WASI functions?

The runtime that is running the code passes the wasi-core functions in as imports.

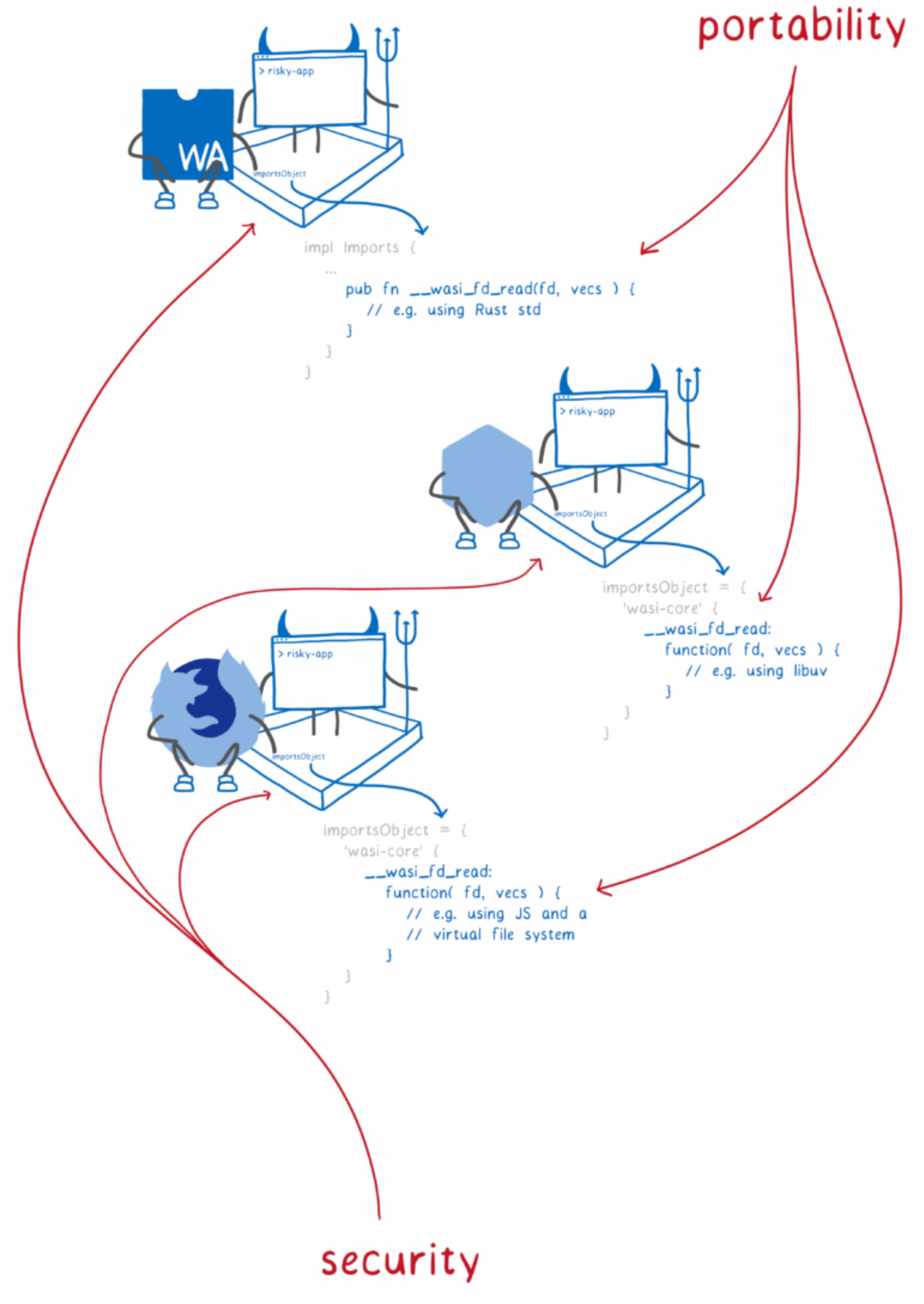

the code the runtime uses to instantiate a module...
...and what it actually does
> risky-app
WA
importsObject

```
instantiate(
    riskyAppBinary,
    importsObject
);
```

```
importsObject = {
  'wasi-core' {
    __wasi_fd_read,
    __wasi_fd_write,
    ...
  }
}
```

This gives us portability, because each host can have their own implementation of wasi-core that is specifically written for their platform—from WebAssembly runtimes like Mozilla's wasmtime and Fastly's Lucet, to Node, or even the browser.

It also gives us sandboxing because the host can choose which wasi-core functions to pass in—so, which system calls to allow—on a program-by-program basis. This preserves security.



```
impl Imports {
  ...
    pub fn __wasi_fd_read(fd, vecs ) {
      // e.g. using Rust std
    }
  }
}
```

```
importsObject = {
  'wasi-core' {
    __wasi_fd_read:
      function( fd, vecs ) {
        // e.g. using libuv
      }
  }
}
```

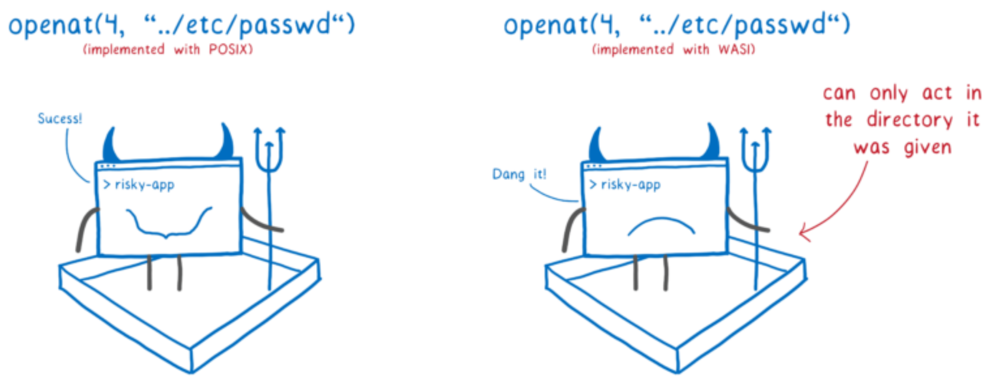
```
importsObject = {
  'wasi-core' {
    __wasi_fd_read:
      function( fd, vecs ) {
        // e.g. using JS and a
        // virtual file system
      }
  }
}
```

WASI gives us a way to extend this security even further. It brings in more concepts from capability-based security.

Traditionally, if code needs to open a file, it calls `open` with a string, which is the path name. Then the OS does a check to see if the code has permission (based on the user who started the program).

With WASI, if you're calling a function that needs to access a file, you have to pass in a file descriptor, which has permissions attached to it. This could be for the file itself, or for a directory that contains the file.

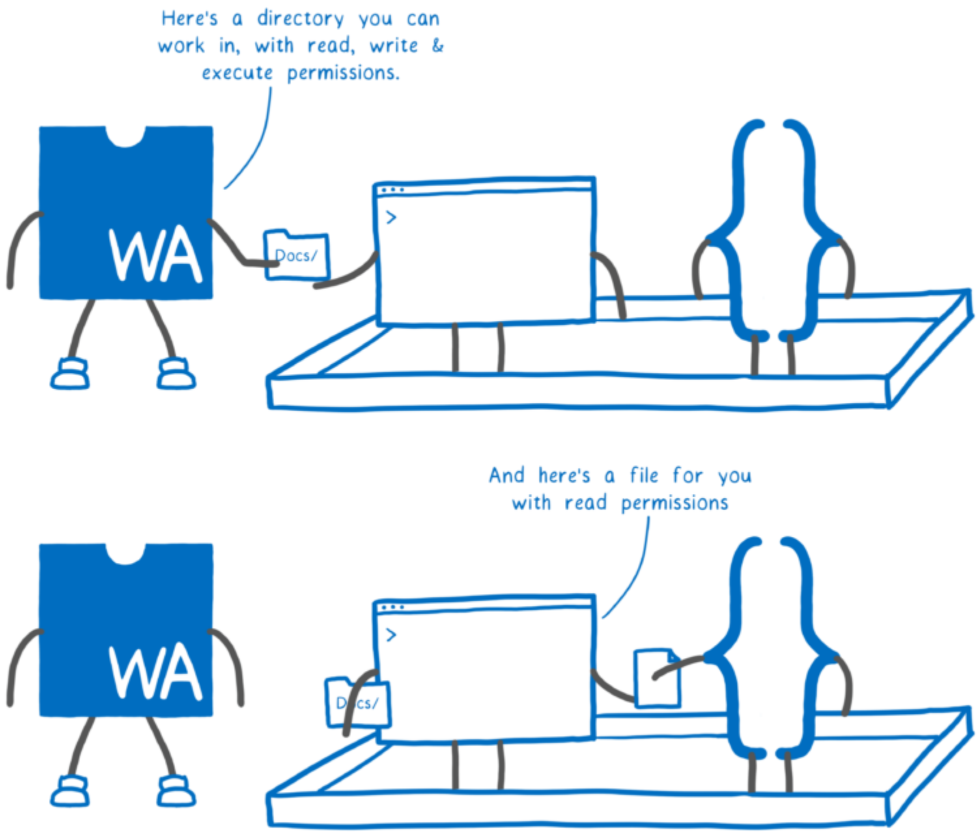
This way, you can't have code that randomly asks to open `/etc/passwd`. Instead, the code can only operate on the directories that are passed in to it.


openat(4, "../etc/passwd")
(implemented with POSIX)
Sucess!
> risky-app
openat(4, "../etc/passwd")
(implemented with WASI)
Dang it!
> risky-app
can only act in the directory it was given

This makes it possible to safely give sandboxed code more access to different system calls—because the capabilities of these system calls can be limited.

And this happens on a module-by-module basis. By default, a module doesn't have any access to file descriptors. But if code in one module has a file descriptor, it can choose to pass that file descriptor to functions it calls in other modules. Or it can create more limited versions of the file descriptor to pass to the other functions.

So the runtime passes in the file descriptors that an app can use to the top level code, and then file descriptors get propagated through the rest of the system on an as-needed basis.


Here's a directory you can work in, with read, write & execute permissions.
WA
Docs/
>
And here's a file for you with read permissions
WA
>
Docs/

This gets WebAssembly closer to the principle of least privilege, where a module can only access the exact resources it needs to do its job.

These concepts come from capability-oriented systems, like CloudABI and Capsicum. One problem with capability-oriented systems is that it is often hard to port code to them. But we think this problem can be solved.

If code already uses `openat` with relative file paths, compiling the code will just work.

If code uses `open` and migrating to the `openat` style is too much up-front investment, WASI can provide an incremental solution. With libpreopen, you can create a list of file paths that the application legitimately needs access to. Then you can use `open`, but only with those paths.

What's next?

We think wasi-core is a good start. It preserves WebAssembly's portability and security, providing a solid foundation for an ecosystem.

But there are still questions we'll need to address after wasi-core is fully standardized. Those questions include:

- asynchronous I/O
- file watching
- file locking

This is just the beginning, so if you have ideas for how to solve these problems, join us!

About Lin Clark

Lin works in Advanced Development at Mozilla, with a focus on Rust and WebAssembly.

https://twitter.com/linclark
@linclark

More articles by Lin Clark…

Learn the best of web development

Sign up for the Mozilla Developer Newsletter:

you@example.com

☐ I'm okay with Mozilla handling my info as explained in this Privacy Policy.

Sign up now

7 comments

Samuel D. Crow

This appears to be a crucial portability hack. 19 years ago Amiga Inc. implemented a product called AmigaDE that did the same thing for Windows and Linux but because it was closed-source the Linux crowd didn't go for it. Now that it's coming back as open-source, I'm glad to see it. I hope it allows alternate OS's the opportunity to crowd out the bloatware that passes for Operating Systems today. I'm especially looking forward to seeing it on AROS and Haiku.

March 27th, 2019 at 10:31

Reply

Duke

I love the concept of wasi.
I'm just wondering: as this is a completely new interface, why keep relying on files, and do not design something with an even simpler, newer abstraction? I've not tought enough about it (so this could be completely stupid), but why instead of "everything is a file" , we do not consider that everything is a monodirectional stream.

this interface has to implement lesser methods maybe it will be more easy to ensure security. On top of this abstraction the app itself could implement helper libraries that simulates file via buffered (or unbuffered) read and write. but as everything is a stream (no seek, just sequential movements) the same abstraction will work for network protocols and maybe others kind of things that an OS usually manage.

I agree this is quite radical and may not fit well with the current way of doing things, but isn't this a new era ? :)

In any case, I love you for doing this! thanks!

March 27th, 2019 at 13:54

Reply

> **Olivier Mengué**
>
> You mean, like Go's io.Reader and io.Writer?
> https://golang.org/pkg/io/#Reader
> https://golang.org/pkg/io/#Writer
>
> March 28th, 2019 at 14:38
>
> Reply

Andy Jackson

Why reinvent the wheel? GoLang (for example) already has a cross-system standard library. So does Node.

Container design is also quite cross-system as well.

That leaves sensor APIs etc. These can continue under the same philosophy.

Or you can just add to a Node API the polyfill APIs of existing browser standards for sensors. Code reuse is always helpful.

Otherwise you risk falling in a hole that's already done.

March 27th, 2019 at 17:02

Reply

Quentin Quaadgras

This is really awesome, wasm looks like it has a promising future ahead of it, it's great to see Mozilla & Rust leading the way on this!

March 28th, 2019 at 02:42

Reply

Roberto Malatesta

Nice, concise and clean article Lin!
Some people like me were waiting for this since Webassembly was announced (and dreamed about that long before, when Emscripten brought in in 2010).

I dare not ask for an ETA though …

— R

PS: FS locking and watching should be top priority. Async I/O is more tricky to adapt on every OS/FS stack, IMHO.

March 28th, 2019 at 02:51

Reply

U007D

+1 to Duke's stream-over-file question. I asked myself the same thing as I read this post–why not a URI-based scheme using streams?

Fantastic writing. And I love the cute graphics–I helps me absorb the teaching through both sides of my brain. I'm sure these postings take a ton of work, but I look forward to them!

Thanks, Lin,
U007D

March 28th, 2019 at 06:37

Reply

Post Your Comment

Your name *

Your e-mail *

Your comment

Submit Comment

Except where otherwise noted, content on this site is licensed under the Creative Commons Attribution Share-Alike License v3.0 or any later version.