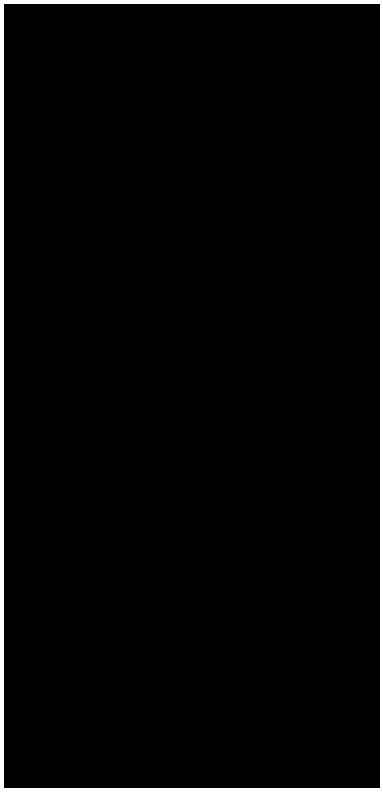
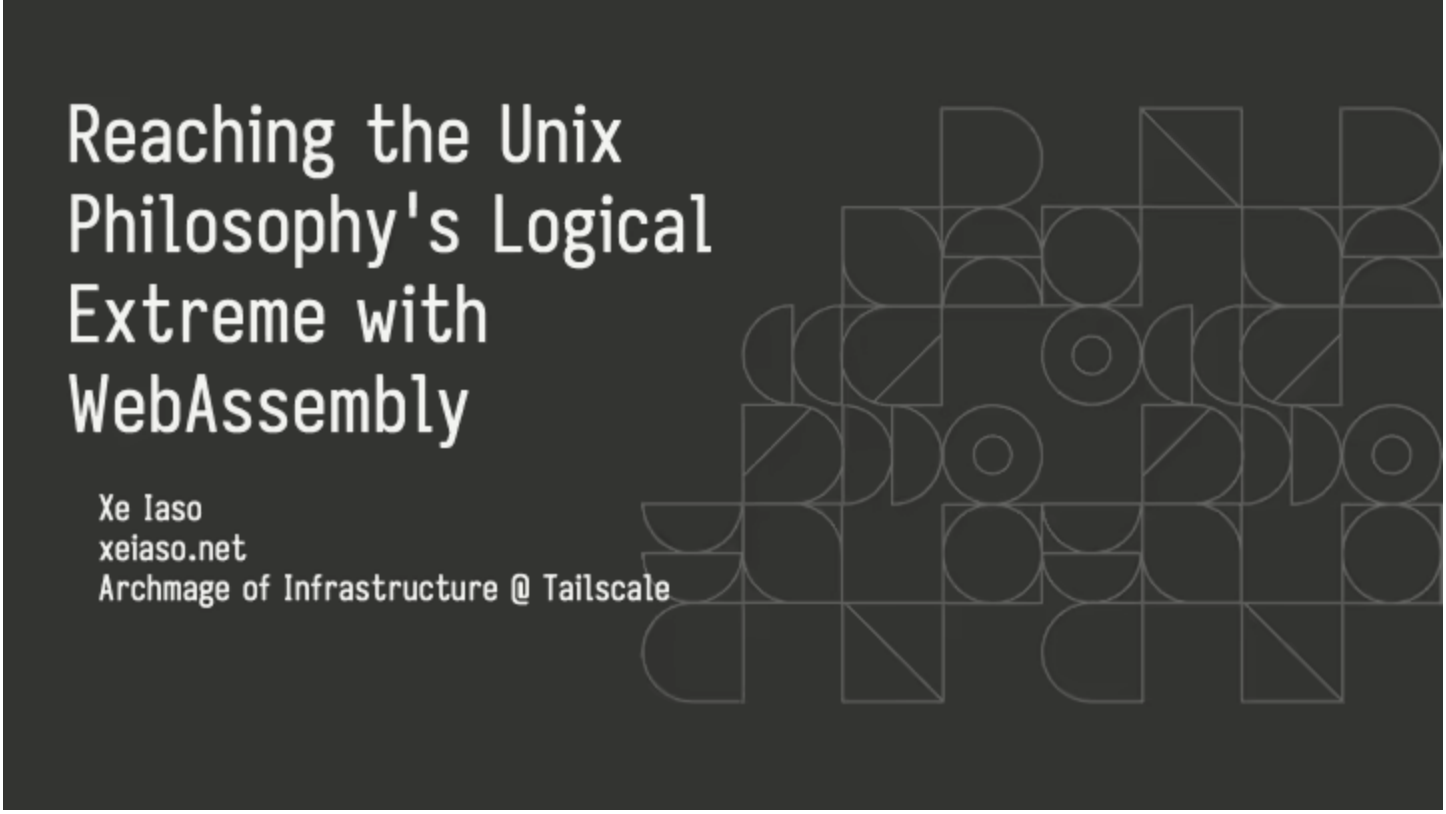
Reaching the Unix Philosophy's Logical Extreme with Webassembly



0:00

YouTube link (please let me know if the iframe doesn't work for you)



Good morning Berlin! How're you doing this fine morning? I'm Xe and today I'm gonna talk about something that I'm really excited about:

WebAssembly. WebAssembly is a compiler target for an imaginary CPU that your phones, tablets, laptops, gaming towers and even watches can run. It's intended to be a level below JavaScript to allow us to ship code in maintainable languages. Today I'm gonna be talking about fun ways you can take advantage of WebAssembly, but first we need to talk about the other main part of this subject:

Unix. Unix is the sole survivor of the early OS wars. It's not really that exciting from a computer science standpoint other than it was where C became popular and it uses file and filesystem API calls to interface with a lot of hardware.

Dealing with some source code files? Discover them in the filesystem in your home directory and write to them with the file API.

Dealing with disks? Discover them in the filesystem and manage them with the file API.

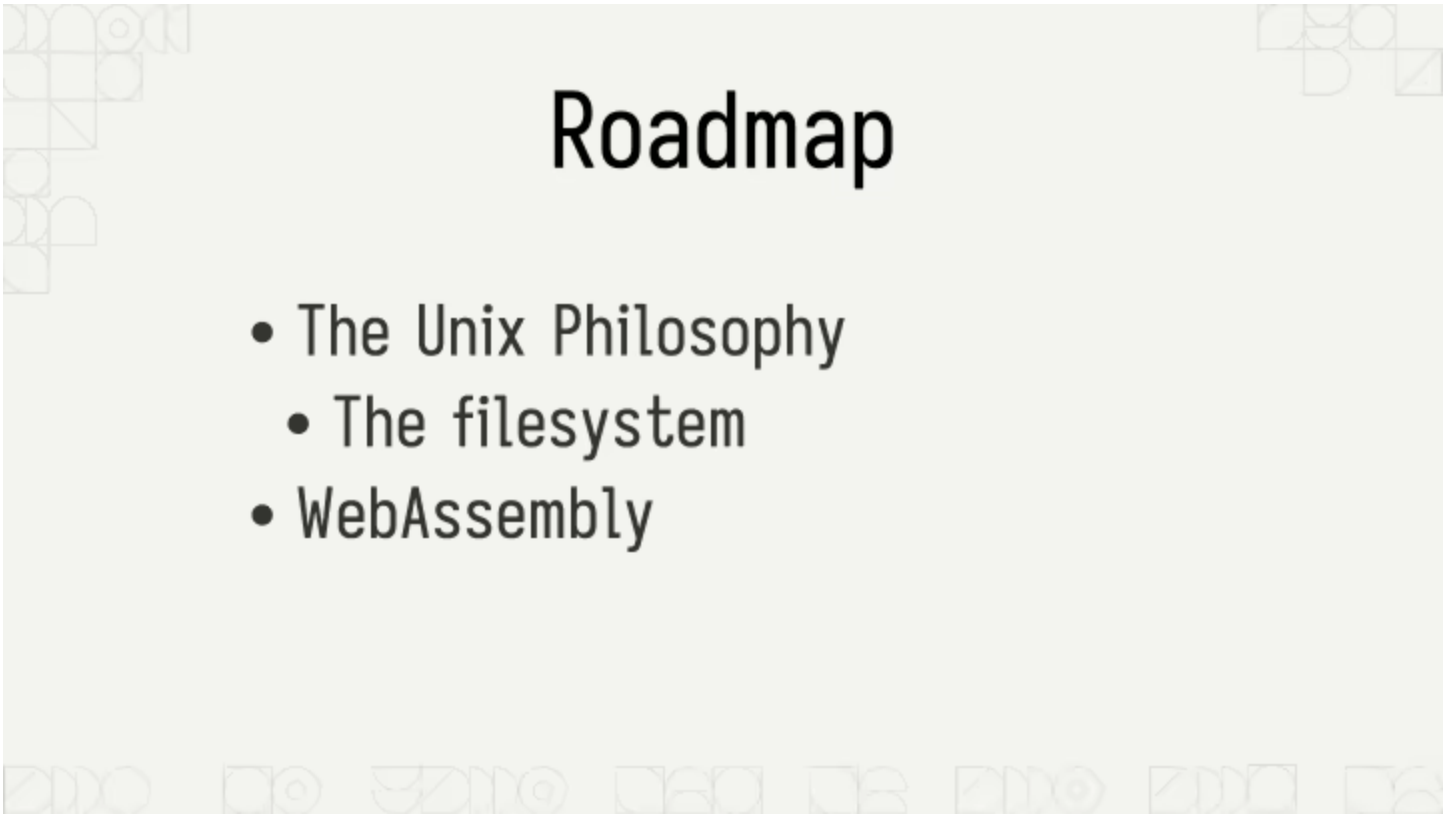
Design is rooted in philosophy, and Unix has a core philosophy that all the decisions stem from. This is usually quoted as "everything is a file" but what does that even mean? How does that handle things that aren't literally files?

(Pause)

And wait, who's this Xe person?

Like the nice person with that microphone said, I'm Xe. I'm the person that put IPv6 packets into S3 and I work at Tailscale doing developer relations. I'm also the only person I know with the job title of Archmage. I'm a prolific blogger and I live in Ottawa in Canada with my husband.

I'm also a philosopher. As a little hint for anyone here, when someone openly introduces themselves as a philosopher, you should know you're in for some fun.



Speaking of fun, I know you got up early for this talk because it sold itself as a WebAssembly talk, but I'm actually going to break a little secret with you. This isn't just a WebAssembly talk. This is an operating systems talk because most of the difficulties with using WebAssembly in the real world are actually operating systems issues. In this talk I'm going to start with the Unix philosophy, talk about how it relates to files, and then I'm gonna circle back to WebAssembly. Really, I promise.

So, going back to where we were with Unix, what does it mean for everything to be a file? What is a file in the first place?

In Unix, what we call "files" are actually just kernel objects we can make a bunch of calls to. And in a very Unix way, file handles aren't really opaque values; they are just arbitrary integers that just so happen to be indices into an array that lives in the struct your kernel uses to keep track of open files in that process. That is the main security model for access to files when running untrusted code in Linux processes.

So with these array indices as arguments to some core system calls you can do some basic calls such as-

(Pause)

Actually, now that I think about it, we just spent half an hour sitting and watching that lovely talk on the Go ecosystem. Let's do a little bit of exercise. Get that blood flowing!

So how many of you can raise your hands? Keep them up, let's get those hands up!

(Pause)

Alright, alright, keep them up.

How many of you have seen one of these 3d printed save icon things in person? If you have, keep your hand up. If not, put it down.

(Pause)

How many of you have used one of them in school, at work, or even at home? If you have, keep it up, if not, put it down.

(Pause)

Alright, thanks again! One more time!

How about one of these audio-only VHS tapes? Keep it up or put it down.

Alright, for those of you with your hands up, it's probably time to schedule that colonoscopy. Take advantage of that socialized medicine! You can put your hands down now, I don't want to be liable.

(Audience laughs)

For the gen-zed in the crowd that had no idea what these things are, a cassette tape was what we used to store music on back when there were 9 planets.

(Audience laughs)

So when I say files, let's think about these. Cassette tapes. Cassette tapes have the same basic usage properties as files.

To start, you can read from files and play music from a cassette tape in all that warm analog goodness. You can write to files and record audio to a cassette tape. Know the term "mixtape"? That's where it comes from. You can also open files and insert a cassette tape into a tape player. When you're done with them, you can close files and remove tapes from a tape player. And finally you can fast-forward and

rewind the tape to find the song you want. Imagine that Gen Z, imagine having to find your songs on the tape instead of skipping right to them.

And these same calls work on log files, hard drives, and more. These 5 basic calls are the heart of Unix that everything spills out from, and this basic model gets you so far that it's how this little OS you've never heard of called Plan 9 works.

But what about things that don't directly map to files? What about network sockets? Network sockets are the abstraction that Unix uses to let applications connect to another computer over a network like the internet. You can open sockets, you can close them, you can read from them, you can write to them. But are they files?

Turns out, they are! In Unix you use mostly the same calls for dealing with network sockets that you do for files. Network sockets are treated like one of these things: an AUX cable to cassette tape adaptor. This was what we used to use in order to get our MP3 players, CD players, Gameboys, and smartphones connected up to the car stereo. This isn't a bit, we actually used these a lot. Yes, we actually used these. I used one extensively when I was delivering pizzas in high school to get the turn by turn navigation directions read out loud to me. We had no other options before Bluetooth existed. It was our only compromise.

(Audience laughs)

How about processes? Those are known to be another hallmark of the Unix philosophy. The Unix philosophy is also understood to be that programs should be filters that take the input and spruce it up for the next stage of the pipeline. Under the hood, are those files?

Yep! Turns out they're three files: input from the last program in the chain, output to the next program in the chain, and error messages to either a log file or operator. All those pipelines in your shell script abominations that you are afraid to touch (and somehow load-bearing for all of production for several companies) become data passing through those three basic files.

It's like an assembly line for your data, every step gets its data fed from the output of the last one and then it sends its results to the input of the next one. Errors might to go an operator or a log sink like the journal, but it goes down the chain and lets you do whatever you want. Really, it's a rather elegant design, there's a reason it's lasted for over 50 years.

So you know how I promised that I'm gonna relate all this back to WebAssembly? Here's when. Now that we understand what Unix is, let's talk about what WebAssembly by itself isn't.

WebAssembly is a CPU that can run pure functions and then return the results. It can also poke the outside world in a limited capacity, but overall it's a lot more like this in practice:

A microcontroller. Sure you can use microcontrollers to do a lot of useful things (especially when you throw in temperature sensors and maybe even a GSM shield to send text messages), but the main thing that microcontrollers can't easily do is connect to the Internet or deal with files on storage devices. Pedantically, this is something you can do, but every time it'll need to be custom-built for the hardware in question. There's no operating system in the mix, so everything needs to have bespoke code. Without an operating system, there's no network stack or even processes. This makes it possible, but moderately difficult to reuse existing code from other projects. If you want to do something like run libraries you wrote in Go on the frontend, such as your peer to peer VPN engine and all of its supporting code, you'd need to either do a pile of ridiculous hacks or you'd just need there to be something close to an operating system. Those hacks would be fairly reliable, but I think we can do better.

(Pause)

Turns out, you don't need an operating system to fill most of the gaps that are left when you don't have one. In WebAssembly, we have something to fill this gap:

WASI. WASI is the WebAssembly System Interface. It defines some semantics for how input, output, files, filesystems, and basic network sockets should be implemented for both the guest program and the host environment. This acts like enough of an "operating system" as far as programming languages care. When you get the Go or Rust compiler to target WASI, they'll emit binaries that can run just about anywhere with a WASI runtime.

<Cadey> I wonder which thing that runs on several billion devices, including the SIM cards in your phones and an alarming number of life-critical devices this reminds you of.

So circling back on the filesystem angle, one of the key distinctions with how WASI implements filesystem access compared to other operating systems is that there's no expectation for running processes to have access to the host filesystem, or even any filesystem at all. It is perfectly legal for a WASI module to run without filesystem access. More critically for the point I'm trying to build up to though, there are a few files that are guaranteed to exist:

Standard input, output, and error files. And, you know what this means? This means we can circle back to the Unix idea of WebAssembly programs being filters. You can make a WebAssembly program take input and emit output as one step in a longer process. Just like your pipelines!

As an example, I have an overcomplicated blog engine that includes its own dialect of markdown because of course it does. After getting nerd sniped by Amos, I rewrote it all in Rust; but when I did that, I separated the markdown parser into its own library and made a little command-line utility for it. I compiled that to WebAssembly with WASI and now I think I'm one of the only people to have successfully distributed a program over the fediverse: the library that I use to convert markdown to HTML, with the furry avatar templates that orange websites hate and all.

Just to help hammer this all in, I'm going to show you some code I wrote between episodes of anime and donghua. I wrote a little "echo server" that takes a line of input, runs a WebAssembly program on that line of input fed into standard in, and then returns the response from standard out. The first program I'm gonna show off is going to be a "reply with the input" program. Then, I'm going to switch it over to my markdown library I mentioned and write out a message to get turned into HTML. I'm going to connect to it

with another WebAssembly program that has a custom filesystem configuration that lets you use the network as a filesystem because WASI's preview 1 API doesn't support making outgoing network connections at the time of writing. If sockets really are just files, then why can't we just use the network stack as a filesystem?

Now, it's time, let's show off the power of WebAssembly. But first, the adequate prayers are needed: Demo gods, hear my cries. Bless my demo!

<Mara> For obvious reasons it's difficult to put the demo into the transcript, but you can find the code for it in github.com/Xe/x.

On the right hand side I have a terminal running that WebAssembly powered echo server I mentioned. Just to prove I didn't prerecord this, someone yell out something for me to type into the WebAssembly program on the left.

(Pause for someone to shout something)

Cool, let's type it in:

(Type it in and hit enter)

See? I didn't prerecord this and that lovely member of the audience wasn't a plant to make this easier on me.

(Audience laughs)

You know what, while we're at it, let's do a little bit more. I have another version of this set up where it feeds things into that markdown->HTML parser I mentioned. If I write some HTML into there:

(Type it in and hit enter)

As you can see, I get the template expanded and all of the HTML goodness has come back to haunt us again. Even though the program on the right is written in Go:

(I press control-backslash to cause the go runtime to vomit the stack of every goroutine, attempting to prove that there's nothing up my sleeve)

It's able to run that Rust program like it's nothing.

(Applause)

Thank Klaus that all that worked. I'm going to put all the code for this on my GitHub page in my x repo.

This technique of embedding Rust programs into Go programs is something I call crabtoespionage. It lets you use the filter property of Unix programs as functions in your Go code. This is how you Rustaceans in crowd can sneak some Rust into prod without having to make sacrifices to the linker gods. I know there's at least one of you out there. Commit the .wasm bytes from rustc or cargo to your repo and then you can still build everything on a Mac, Plan 9, or even TempleOS, assuming you have Go running there.

Most of the heavy lifting in my examples is done with Wazero, it's a library for Go programs that is basically a WebAssembly VM and some hooks for WASI implemented for you. The flow for embedding Rust programs into Go looks like this:

- First, extract the subset of the library you want and make it a standalone program. This makes it easy to test things on the command line. Use arguments and command line flags, they're there for a reason.
- Next, build that to WASI and fix things until it works. You'll have to figure out how to draw the rest of the owl here. Some things may be impossible depending on facts and circumstances. Usually things should work out.
- Then import wazero into your program and set everything up by using the embed directive to hoist the WebAssembly bytes into your code. Set up the filesystems you want to use, and your runtime config and finally:
- Then make a wrapper function that lets you go from input to output et voila!

You've just snuck Rust into production. This is how I snuck Rust into prod at work and nobody is the wiser.

(Pause)

Wait, I just gave it away, no, oops. Sorry! I had no choice. Mastodon HTML is weird. The Go HTML library is weirder.

There's a few libraries on GitHub that use this basic technique for more than just piping input to output, they use it to embed C and C++ libraries into Go code. In the case of the regular expressions package, it can be faster than package regexp in some cases. Including the WebAssembly overhead. It's incredible. There's not even that many optimizations for WebAssembly yet!

No C compiler required! No cross-compiling GCC required! No satanic sacrifices to the dark beings required! It's magic, just without the spell slots.

So while we're on this, let's take both aspects of this to their logical conclusions. What about plugins for programs? There's plenty of reasons customers would want to have arbitrary plugin code running, and also plenty of reasons for you to fear having to run arbitrary customer plugin code. If we can run things in an isolated sandbox and then define our own filesystem logic: what if we expose application state as a filesystem? Trigger execution of the plugin code based on well-defined events that get piped to standard input. Make open calls fetch values from an API or write new values to that same API.

This is how the ACME editor for Plan 9 works. It exposes internal application state as a filesystem for plugins to manipulate to their pleasure.

So, to wrap all of this up:

- When you're dealing with Unix, you're dealing with files, be they source code, hard drives, or network sockets.
- Like anything made around a standards body, even files themselves are lies and anything can be a file if it lies enough in the right way.
- Understanding that everything is founded on these lies frees you from the expectation of trying to stay consistent with them. This lets you run things wherever without having to have a C compiler toolchain for win32 handy.
- Because everything is based on these lies, if you control what lies are being used, you actually end up controlling the truths that users deal with. When you free yourself from the idea of having to stay consistent with previous interpretation of those lies, you are free to do whatever you want.

How could you use this in your projects to do fantastic new things? The ball's in your court Creators.

(Applause)

With all that said, here's a giant list of everyone that's helped me with this talk, the research I put into the talk, and uncountable other things. Thank you so much everyone.

(Thunderous applause)

And with that, I've been Xe! Thank you so much for coming out to Berlin. I'll be wandering around if you have any questions for me, but if I miss it, please do email me at crabtoespionage@xeserv.us. I'll reply to your questions, really. My example code is in the conferences folder of my experimental repo github.com/Xe/x. Otherwise, please make sure to stay hydrated and enjoy the conference! Be well!

Link to the slides

▶ Share on Mastodon

This talk was posted on M08 27 2023. Facts and circumstances may have changed since publication Please contact me before jumping to conclusions if something seems wrong or unclear.

The art for Mara was drawn by Selicre.

The art for Cadey was drawn by ArtZora Studios.

Some of the art for Aoi was drawn by @Sandra_Thomas01.

Like what you see? Donate on Patreon like these awesome people!

Looking for someone for your team? Take a look here.

See my salary transparency data here.

Served by /nix/store/c5y9p9c0nwngisq59ma5vdav9w3i8hf5-xesite-3.0.0/bin/xesite, see source code here.