

Using WebAssembly to run, extend and secure your app

Niels Tanis
Sr. Principal Security Researcher

VERACODE



Who am I?



- Niels Tanis
- Sr. Principal Security Researcher
 - Background .NET Development, Pentesting/ethical hacking, and software security consultancy
 - Research on static analysis for .NET apps
 - Enjoying Rust!
- Microsoft MVP - Developer Technologies

VERACODE

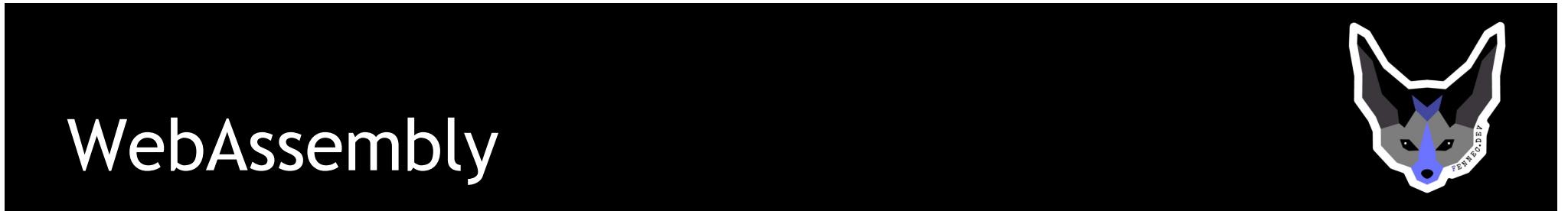


@niels.fennec.dev @nielstanis@infosec.exchange

Agenda



- Introduction
- WebAssembly 101
- Running on WebAssembly
- Extending with WebAssembly
- Securing with WebAssembly
- Conclusion
- Q&A



The screenshot shows the official WebAssembly website at https://webassembly.org. The page has a dark header with the title 'WebAssembly'. Below the header is a navigation bar with links for Overview, Getting Started, Specs, Feature Extensions, Community, and FAQ. A main heading 'WEBASSEMBLY' is followed by a sub-section stating 'WebAssembly 1.0 has shipped in 4 major browser engines.' with icons for Firefox, Chrome, Safari, and Edge. A descriptive paragraph explains what WebAssembly is, mentioning it's a binary instruction format for a stack-based virtual machine designed for portable compilation across client and server applications. A yellow callout box at the bottom provides developer documentation details.

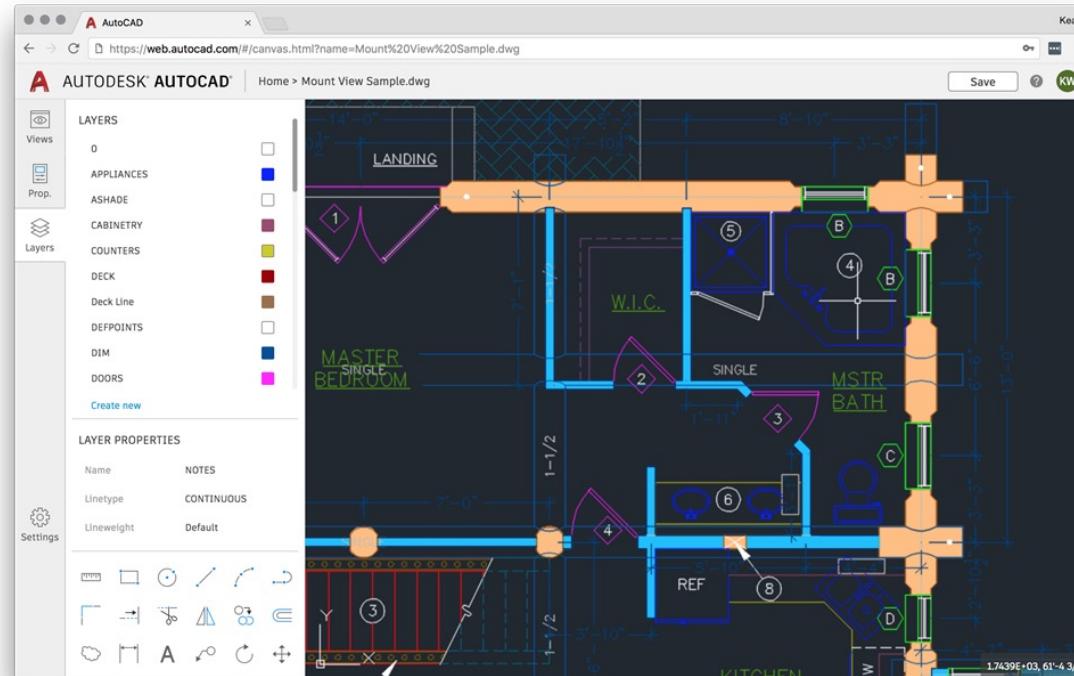
Developer reference documentation for Wasm can be found on [MDN's WebAssembly pages](#). The open standards for WebAssembly are developed in a [W3C Community Group](#) (that includes representatives from all major browsers) as well as a [W3C Working Group](#).



 @niels.fennec.dev  @nielstanis@infosec.exchange



WebAssembly - AutoCAD



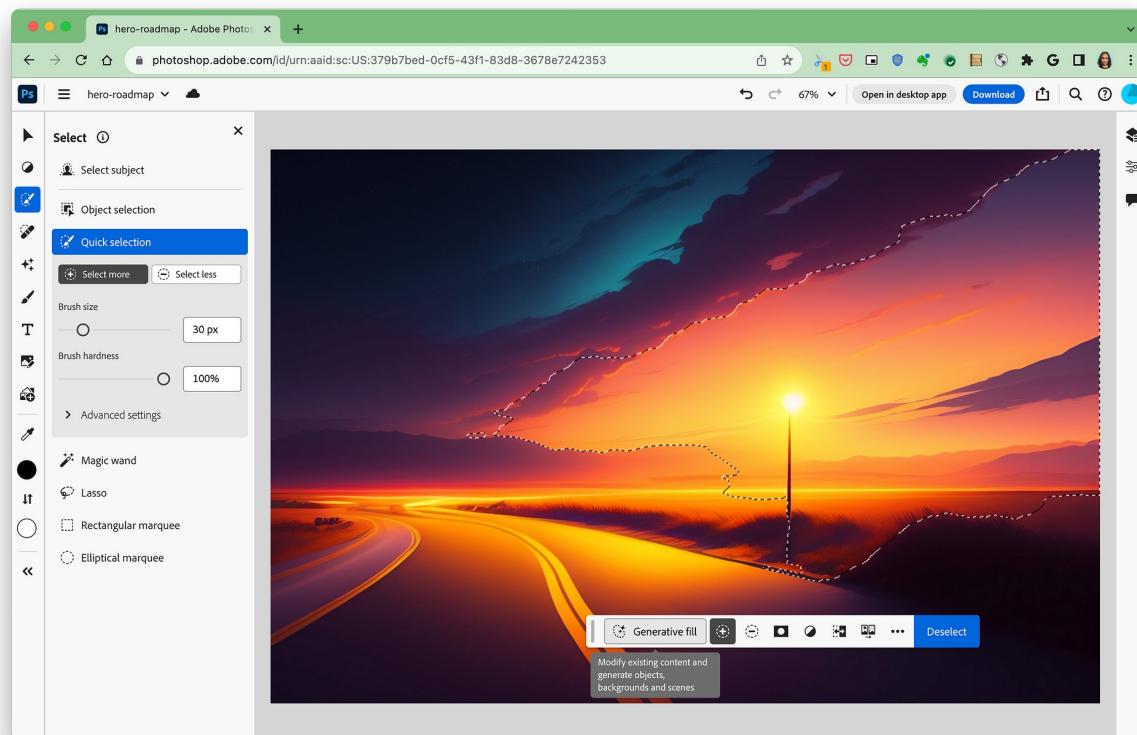
@niels.fennec.dev



@nielstanis@infosec.exchange



WebAssembly - Photoshop



🔗 [@niels.fennec.dev](https://niels.fennec.dev) 📠 [@nielstanis@infosec.exchange](https://nielstanis.infosec.exchange)



WebAssembly - SDK's

A screenshot of a Medium article page. The title is "Introducing the Disney+ Application Development Kit (ADK)" by Mike Hanley. The article was published on Sep 8, 2021, and has a 10 min read time. It features a photo of Mike Hanley and social sharing icons. The content discusses the Disney+ Application Development Kit (ADK) and its benefits for developers.

A screenshot of an Amazon Science article. The title is "How Prime Video updates its app for more than 8,000 device types" by Alexandru Ene. The article is dated January 27, 2022, and mentions the switch to WebAssembly for increased stability and speed. It also notes that Prime Video delivers content to millions of customers on various devices.



↗ @niels.fennec.dev ↗ @nielstanis@infosec.exchange



Prime Video UI - Rust & WASM

The screenshot shows a web browser window with the following details:

- Title Bar:** Rebuilding Prime Video UI with Rust & WebAssembly
- URL:** www.infoq.com/presentations/prime-video-rust/
- Video Player:** A video player showing the first slide of the presentation. The slide has a dark blue background with the InfoQ logo at the top and the title "REBUILDING THE PRIME VIDEO UI WITH RUST AND WEBASSEMBLY" in white. Below the title, it says "ALEXANDRUENE PRINCIPAL ENGINEER @ PRIME VIDEO". The video duration is 48:06.
- Controls:** Standard video controls like play, volume, and progress bar.
- Buttons:** Download, SLIDES, and 48:06.
- Summary Section:** A summary of the talk: "Alexandru Ene features details of a new UI SDK in Rust for Prime Video that targets living room devices."
- About the conference:** A section about QCon San Francisco, stating: "Software is changing the world. QCon San Francisco empowers software development by facilitating the spread of knowledge and innovation in the developer community."
- Transcript:** A transcript of the video, starting with: "Ene: We're going to talk about how we rebuilt a Prime Video UI for living room devices with Rust and WebAssembly, and the journey that got us there. I'm Alex. I've been working on this project with a team of developers from Amazon Prime Video."
- Recorded at:** QCon SAN FRANCISCO by InfoQ
- Date:** MAR 21, 2025



WebAssembly - Do you use it?



- Does your organization use WASM?
- Which main tech stacks used?
- WASM for what kind of apps?
 - Mobile
 - Cloud Native
 - Other...



WEBASSEMBLY

WebAssembly Design



- **Be fast, efficient, and portable**
 - Executed in near-native speed across different platforms
- **Be readable and debuggable**
 - In low-level bytecode but also human readable
- **Keep secure**
 - Run on sandboxed execution environment
- **Don't break the web**
 - Ensure backwards compatibility

WebAssembly



- Binary instruction format for stack-based virtual machine similar to .NET CLR running MSIL or JVM running bytecode
- Designed as a portable compilation target
- The security model of WebAssembly:
 - Protect users from buggy or malicious modules
 - Provide developers with useful primitives and mitigations for developing safe applications



WebAssembly Type System

- WebAssembly's type only supports:
 - i32 (32-bit integer)
 - i64 (64-bit integer)
 - f32 (32-bit float)
 - f64 (64-bit float)
- No strings, no objects, no complex data types.
- Basic operations on numerical values.

WebAssembly Stack Based VM



- In a stack-based VM, operations primarily manipulate a last-in-first-out (LIFO) stack of values, rather than working with named registers as in register-based architectures.



WebAssembly Stack Based VM

- WebAssembly maintains an operand stack during execution:
 - Values are pushed onto the stack by instructions
 - Operations pop their operands from the stack
 - Results are pushed back onto the stack
 - The stack is separate from linear memory and is not directly accessible

Code some WebAssembly



Let's create some WAT!

<http://github.com/niestanis/secappdev25wasm>



WEBASSEMBLY

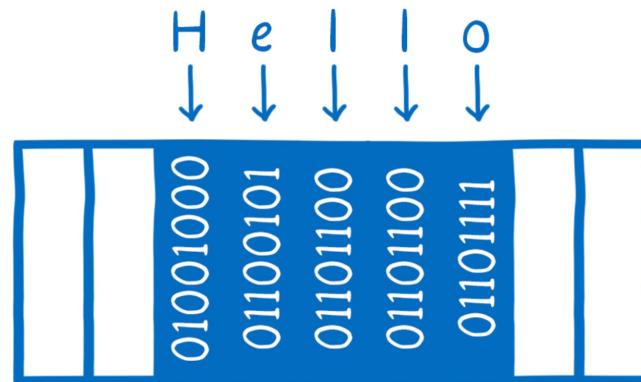


 @niels.fennec.dev  @niestanis@infosec.exchange



WebAssembly Memory

- Isolated per WASM module
- A contiguous, mutable array of uninterpreted bytes



Code some WebAssembly



Let's work with some Memory!



WEBASSEMBLY



 @niels.fennec.dev  @nielstanis@infosec.exchange

WebAssembly Control-Flow Integrity



- Control-Flow Integrity (CFI) ensures that program execution follows only valid paths as defined by the program's source code.
- In traditional native code, attackers can exploit memory vulnerabilities to hijack execution flow:
 - Redirecting it to malicious code
 - Chaining together existing code fragments in unintended ways (like Return-Oriented Programming attacks).



CFI Example

```
int number = Convert.ToInt32(Console.ReadLine());
Console.WriteLine($"Number {number}");
if (number>5)
{
    Console.WriteLine("Number is larger than 5");
}
else
{
    Console.WriteLine("Number is smaller than 5");
}
Console.WriteLine("Done!");
```



🔗 [@niels.fennec.dev](https://niels.fennec.dev) 📩 [@nielstanis@infosec.exchange](mailto:nielstanis@infosec.exchange)



CFI Example

```
int number = Convert.ToInt32(Console.ReadLine());  
Console.WriteLine($"Number {number}");  
if (number>5)
```

```
Console.WriteLine("Number is larger than 5");
```

```
Console.WriteLine("Number is smaller than 5");
```

```
Console.WriteLine("Done!");
```



CFI: Structured Control-Flow Only

- No arbitrary jump instructions (no goto, jmp or equivalent)
- All branches have explicit, validated targets within the same function
- No ability to jump to arbitrary memory addresses
- All control flow is validated before execution



CFI: Protected Function Calls

```
(module
  (func $safe_function (param i32) (result i32)
    local.get 0
    i32.const 1
    i32.add
  )

  (func $caller (param i32) (result i32)
    local.get 0
    call $safe_function ; Direct call - target validated at compile time
  )
)
```



CFI: Indirect Calls

- Function table entries are validated at module instantiation
- Runtime checks ensure the target index is within table bounds
- Runtime type checking ensures the called function matches the expected signature
- The table is protected from direct memory manipulation

Code some WebAssembly



Indirect calling!



WEBASSEMBLY



 @niels.fennec.dev  @nielstanis@infosec.exchange



CFI: Indirect Calls

```
(module
  (table 2 funcref)
  (func $f1 (result i32)
    i32.const 42)
  (func $f2 (result i32)
    i32.const 13)
  (elem (i32.const 0) $f1 $f2)
  (type $return_i32 (func (result i32)))
  (func (export "callByIndex") (param $i i32) (result i32)
    local.get $i
    call_indirect (type $return_i32))
)
```



CFI: Separation of Code and Data

- Code sections cannot be modified at runtime
- Linear memory (accessible data) cannot contain executable code
- No instruction exists to convert data to code No self-modifying code capabilities

RLBox



RLBox



@niels.fennec.dev



@nielstanis@infosec.exchange



RLBox in Firefox

- RLBox uses **WebAssembly** as an **intermediate compilation target** for creating sandboxed versions of third-party C/C++ libraries.
- Rather than running WebAssembly code directly, RLBox employs a unique "**WebAssembly and Back Again**" approach.

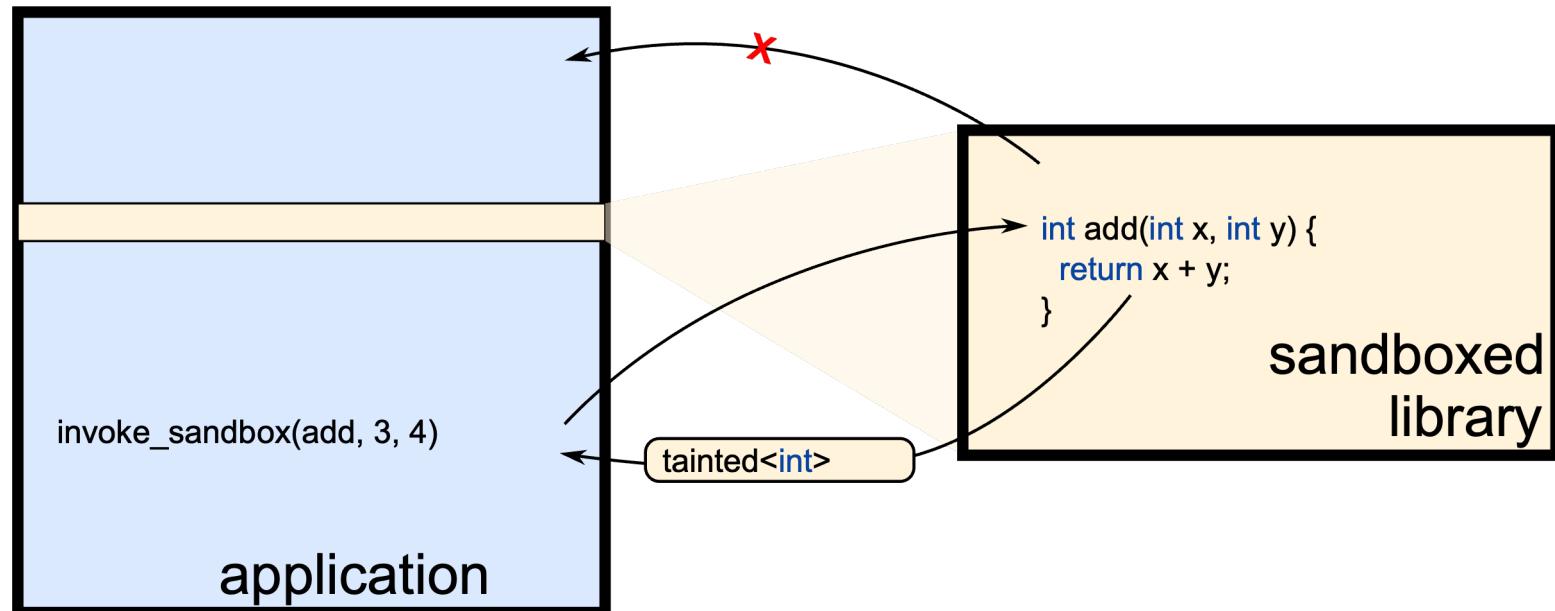


RLBox in Firefox

- RLBox considers all values that originate in the sandbox as untrusted and "taints" them.
- Tainted values are essentially opaque values that cannot be used directly by the application code.
- RLBox's type system marks all data coming out of the sandbox as "tainted" and ensures, through compiler errors, that developers sanitize potentially unsafe data before using it.



RLBox in Firefox



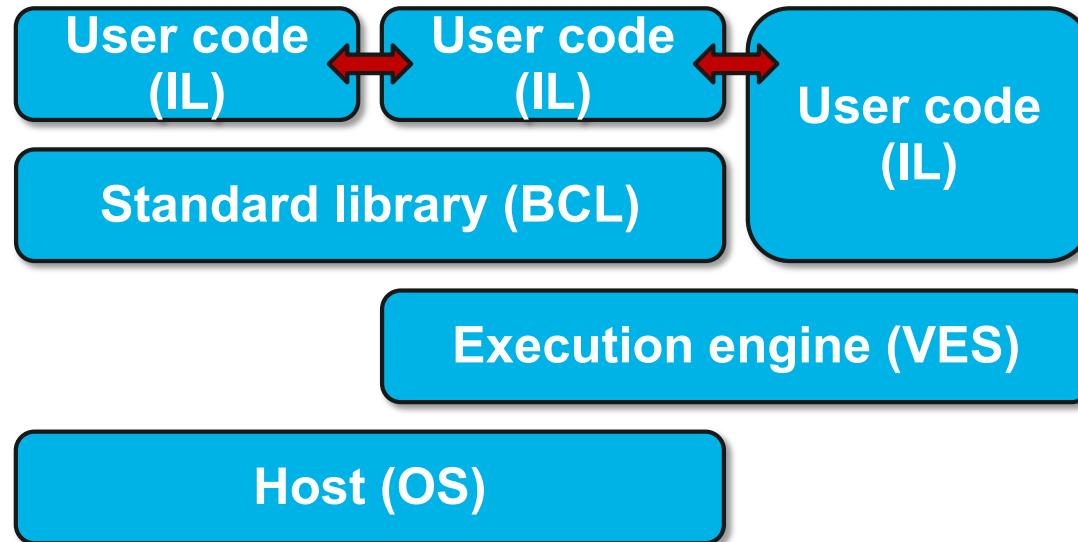


RLBox in Firefox

- RLBox is currently deployed in Firefox to isolate five different modules:
 - Graphite - Font rendering engine - Firefox 95
 - Hunspell - Spell checker - Firefox 95
 - Ogg - Multimedia container format - Firefox 95
 - Expat - XML parser - Firefox 96
 - Woff2 - Web font compression format - Firefox 96

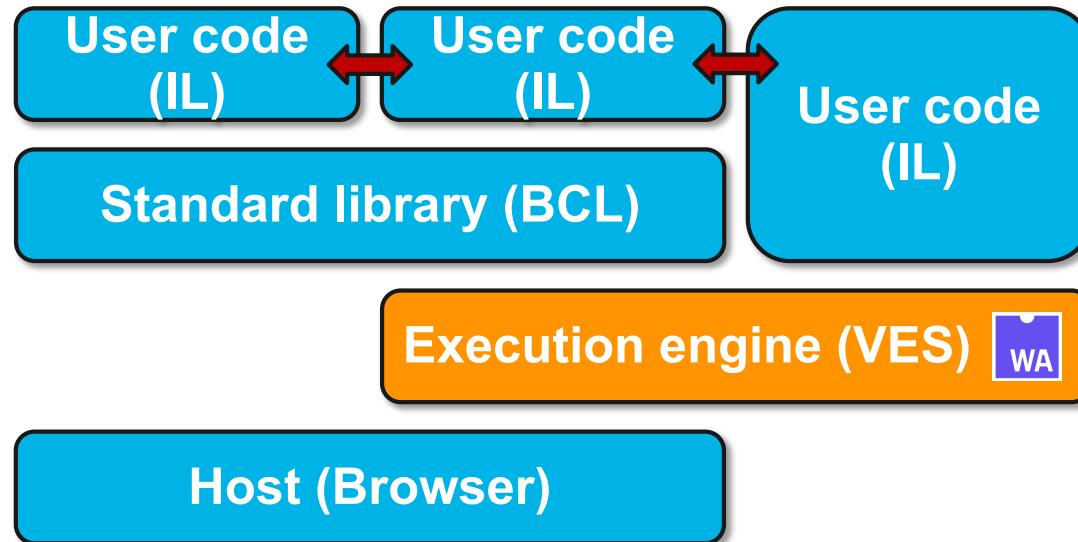


Running .NET on WebAssembly



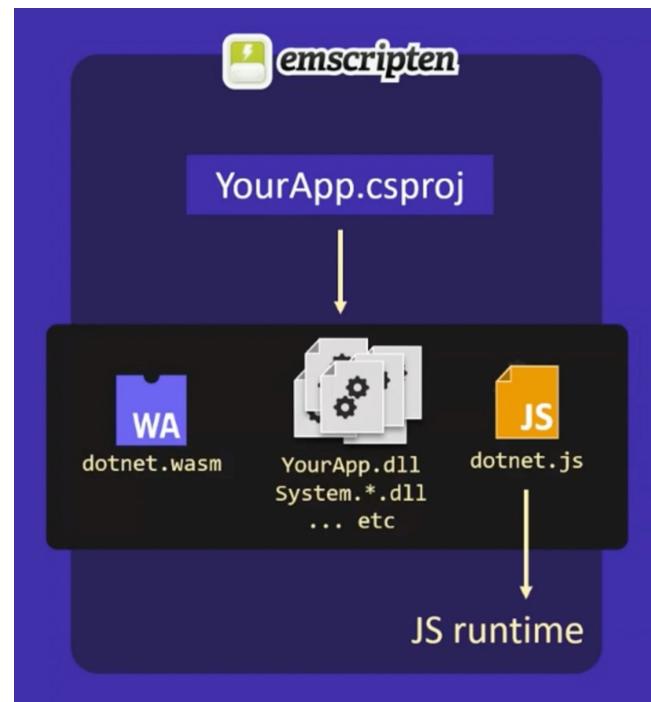


Running .NET on WebAssembly





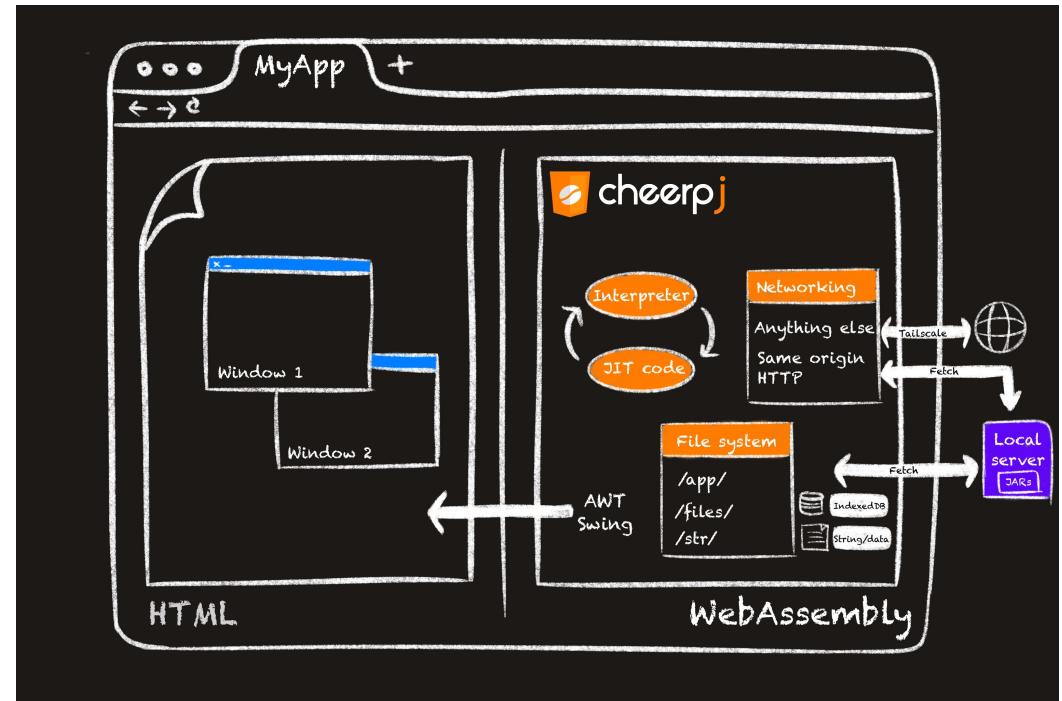
Blazor WebAssembly





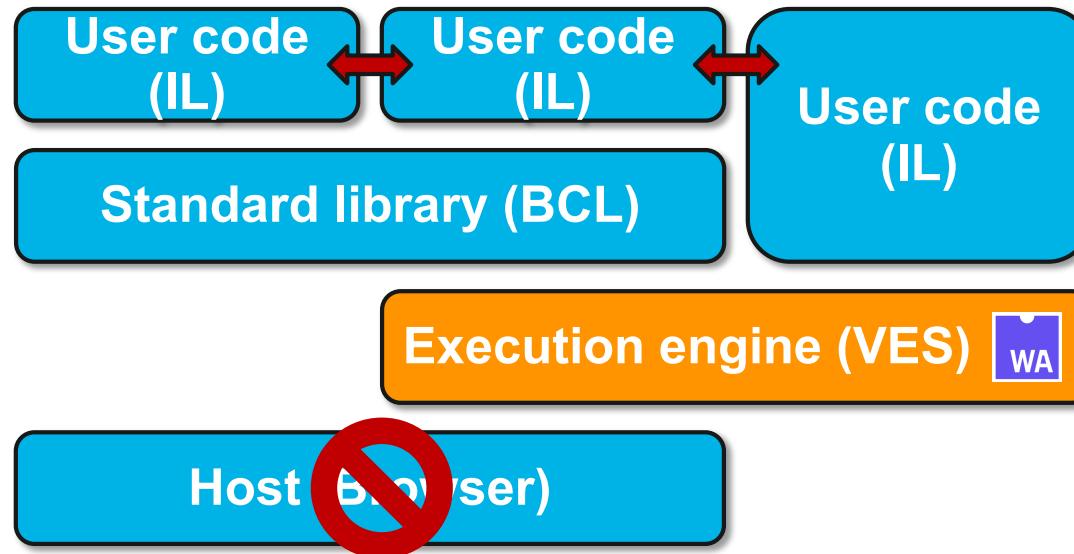
JVM on WASM with CheerpJ

The magic behind CheerpJ is Cheerp, which was used for compiling full Java SE 8 and Java SE 11 runtimes based on OpenJDK.





Running .NET on WebAssembly



WebAssembly System Interface WASI



- Introduced in March 2019 by Bytecode Alliance
- WasmTime implementation as reference
- POSIX inspired, engine-independent, non-Web system-oriented API for WebAssembly

WebAssembly System Interface WASI



- Strong sandbox with Capability Based Security
- Right now, supports e.g. FileSystem actions, Sockets, CLI and HTTP at version 0.2
- Future support for promise/async and streams
- Anyone recall .NET Standard? ☺



Docker vs WASM & WASI

Solomon Hykes op Twitter: "If I had to choose between Docker and WASM/WASI, I'd choose WASM/WASI every time. 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 · 27 mrt. 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/standalone-wasi/



Docker vs WASM & WASI

A screenshot of a Twitter tweet from Solomon Hykes (@solomonstre). The tweet reads: "So will wasm replace Docker?" No, but imagine a future where Docker runs linux containers, windows containers and wasm containers side by side. Over time wasm might become the most popular container type. Docker will love them all equally, and run it all :)". Below the main tweet is a reply from Solomon Hykes (@solomonstre) dated 27 mrt. 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! twitter.com/linclark/status...". The tweet has 56 Retweets, 5 Geciteerde Tweets, and 165 Vind-ik-leuks.



Docker & WASM

The screenshot shows a web browser window with the title "Introducing the Docker+Wasm Technical Preview". The page features the Docker+Wasm logo at the top left. The main heading is "Introducing the Docker+Wasm Technical Preview". Below the heading is a bio for Michael Irwin, featuring a small profile picture, the name "MICHAEL IRWIN", and the date "Oct 24 2022". A note at the bottom states: "The Technical Preview of Docker+Wasm is now available! Wasm has been producing a lot of buzz recently, and this feature will make it easier for you to quickly build applications targeting Wasm runtimes." To the right of the browser window is a diagram titled "Docker Engine" which shows the internal architecture of Docker+Wasm. It illustrates how the Docker Engine interacts with containerd, which then manages multiple runtime environments: containerd-shim (using runc to run Container process), containerd-shim (using runc to run Container process), and containerd-wasm-shim (using wasmedge to run Wasm Module).

Wasm is a fast, light alternative to Linux containers — try it out today in the [Docker+Wasm Technical Preview](#)

MICHAEL IRWIN
Oct 24 2022

The Technical Preview of Docker+Wasm is now available! Wasm has been producing a lot of buzz recently, and this feature will make it easier for you to quickly build applications targeting Wasm runtimes.

Docker Engine

containerd

containerd-shim

runc

Container process

containerd-shim

runc

Container process

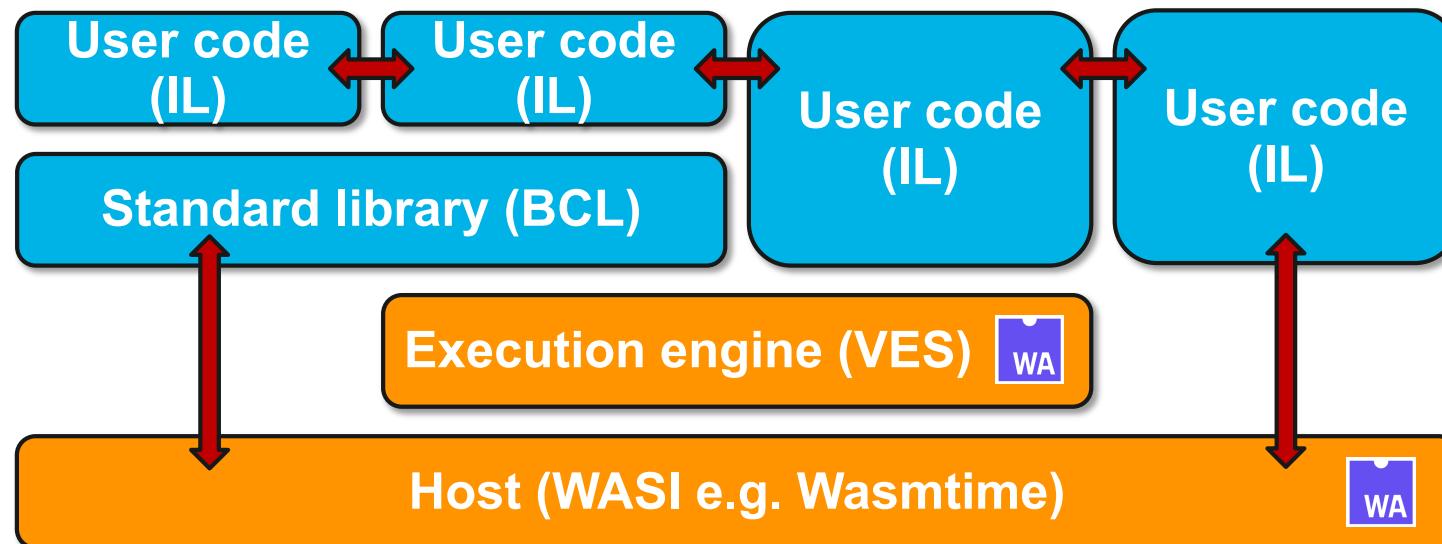
containerd-wasm -shim

wasmedge

Wasm Module



WebAssembly System Interface WASI





Experimental WASI SDK for .NET





.NET 8 WASI-Experimental

A screenshot of a Microsoft Edge browser window displaying a Microsoft Dev Blogs post. The title of the post is "Extending WebAssembly to the Cloud". The main content discusses the ".NET 8 WASI-Experimental workload". It explains that .NET 8 includes a new workload called "wasi-experimental" which builds on Wasm functionality used by Blazor, extending it to run in "wasmtime" and invoke WASI interfaces. The post provides instructions for installing the workload via the command "dotnet workload install wasi-experimental". It also notes that admin permissions may be required and that "wasmtime" needs to be installed. A code snippet shows how to create a new "wasiconsole" application.

wasi-experimental workload

.NET 8 includes a new workload called [wasi-experimental](#). It builds on top of the Wasm functionality used by Blazor, extending it to run in [wasmtime](#) and invoke WASI interfaces. It is far from done, but already enables useful functionality.

Let's move on from theory to demonstrating the new capabilities.

After installing the [.NET 8 SDK](#), you can install the [wasi-experimental](#) workload.

```
dotnet workload install wasi-experimental
```

Note: This command may require admin permissions, for example with [sudo](#) on Linux and macOS.

You also need to install [wasmtime](#) to run the Wasm code you are soon going to produce.

Try a simple example with the [wasi-console](#) template.

```
$ dotnet new wasiconsole -o wasiconsole
$ cd wasiconsole
$ cat Program.cs
using System;

Console.WriteLine("Hello, WASI Console!");
```

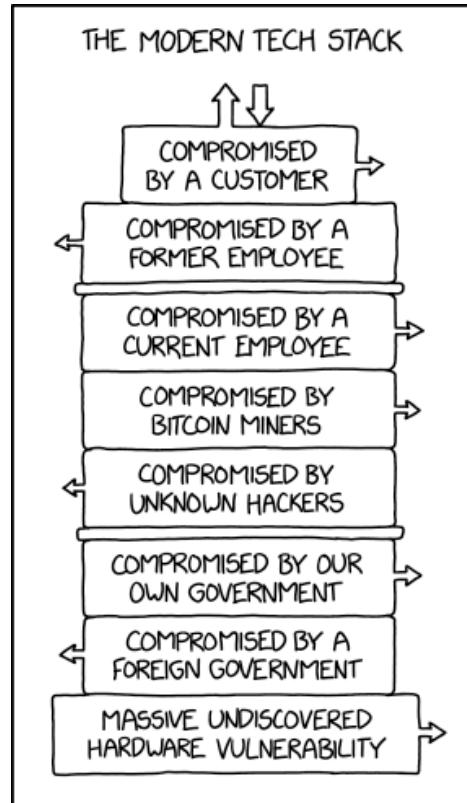


Extending .NET with WASM

- WasmTime.NET NuGet package
- Can run WASM inside of any .NET application
- Extend with Rust based WASM module
- Limit capabilities
- Demo time!



Trusted Computing - XKCD 2166





Enarx

The screenshot shows the Enarx website homepage. At the top, there's a navigation bar with links for Enarx, Docs, Resources, Community, Initiatives, Star, and Search. Below the navigation is a large header with the Enarx logo and the text "Confidential Computing with WebAssembly". There are two prominent buttons: "Download" and "Try Enarx". The main content area features three sections: "100% Open Source" (with an icon of a padlock), "Easy Deployment" (with an icon of a box being opened), and "Cloud Native, Hardware Neutral" (with an icon of a central processing unit). Each section contains a brief description of Enarx's capabilities.

Enarx

Confidential Computing with WebAssembly

[Download](#) [Try Enarx](#)

100% Open Source

Enarx is the leading open source framework for running applications in TEEs (Trusted Execution Environments). It's part of the Confidential Computing Consortium from the Linux Foundation.

Easy Deployment

Enarx provides a run-time TEE based on WebAssembly, allowing developers to deploy applications without any rewrites from languages like Rust, C/C++, C#, Go, Java, Python, Haskell and many more.

Cloud Native, Hardware Neutral

Enarx is CPU-architecture independent, letting developers deploy the same application code transparently across multiple targets. It provides a single runtime and attestation framework which is hardware vendor and CSP neutral.



@niels.fennec.dev

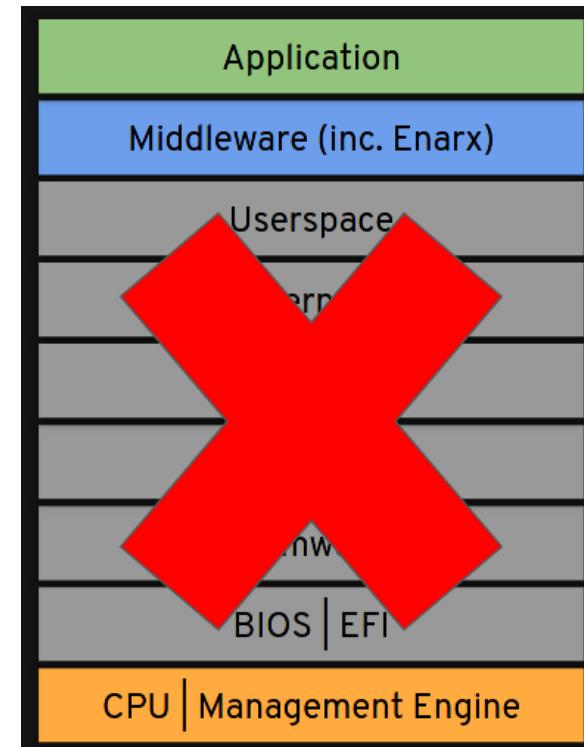


@nielstanis@infosec.exchange



Enarx Threat Model

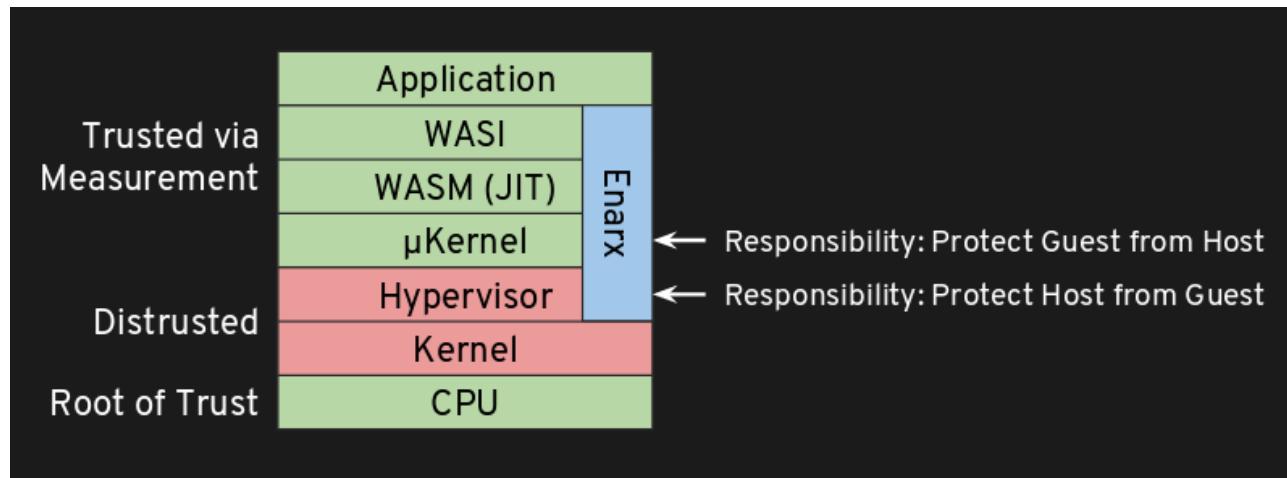
- Don't trust the host
- Don't trust the host owner
- Don't trust the host operator
- Hardware cryptographically verified
- Software audited and cryptographically verified





Enarx

- Leverages Trusted Execution Environment (TEE) direct on processor
 - AMD's SEV, Intel's SGX and IBM's PEF
- Attestation of hardware and Enarx runtime



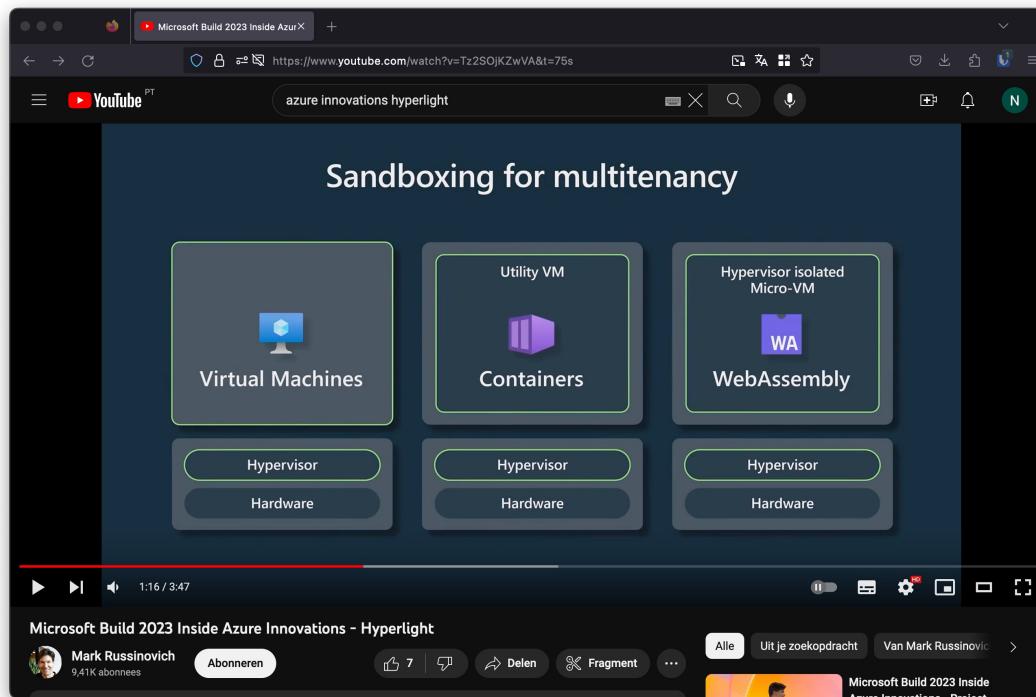
Enarx



- Each execution:
 - Attestation: Enarx checks that the host to which you're planning to deploy is a genuine TEE instance.
 - Packaging: Once the attestation is complete and the TEE instance verified, the Enarx management component encrypts the application, along with any required data.
 - Provisioning: Enarx then sends the application and data along to the host for execution in the Enarx Keep.
- TEE provides: Data Confidentiality, Data Integrity, Code Integrity



Project Hyperlight



@niels.fennec.dev



@nielstanis@infosec.exchange



DotNetIsolator & Project Hyperlight

YouTube video player showing a man speaking about DotNetIsolator. A purple overlay box contains the text ".NET WA .NET".

DotNetIsolator: an experimental package for running .NET code in an isolated sandbox

YouTube video player showing a presentation titled "Sandboxing Your Sandbox: Leveraging Hypervisors for WebAssembly Security" by Danilo (Dan) Chiarlone at WASMCON.

Sandboxing Your Sandbox: Leveraging Hypervisors for WebAssembly Security
By: Danilo (Dan) Chiarlone

WASMCON BETTER TOGETHER



@niels.fennec.dev @nielstanis@infosec.exchange



Hyperlight CNCF Sandbox

The screenshot shows a Microsoft Edge browser window with a dark theme. The address bar reads "Hyperlight Wasm: Fast, secure, opensource.microsoft.com/blog/2025/03/26/hyperlight-wasm-fast-secure-and-". The main content area features a large title "Hyperlight Wasm: Fast, secure, and OS-free" in white. Below the title, it says "Project updates • March 26, 2025 • 10 min read". The article is written by "Yosh Wuyts, Senior Developer Advocate" and "Lucy Menon, Software engineer and researcher, Microsoft". To the right of the text is a photograph of two people, a man and a woman, looking at a computer screen together. The bottom of the page has a "SHARE" section with icons for Facebook, Twitter, and LinkedIn, and a "CONTENT TYPE" section.

Project updates • March 26, 2025 • 10 min read

Hyperlight Wasm: Fast, secure, and OS-free

By [Yosh Wuyts](#), Senior Developer Advocate
Lucy Menon, Software engineer and researcher, Microsoft

SHARE

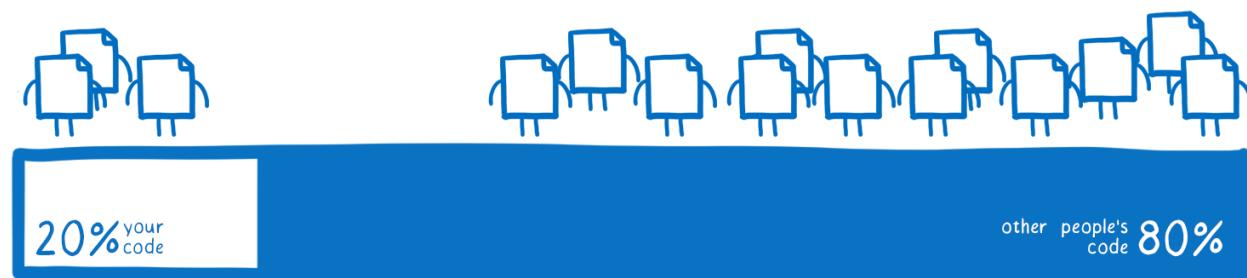
CONTENT TYPE

Last fall the Azure Core Upstream team [introduced Hyperlight](#): an open-source Rust library you can use to execute small, embedded functions using hypervisor-based protection. Then, we showed how to run [Rust functions really, really fast](#), followed by using [C to run Javascript](#). In February 2025, the Cloud Native Computing Foundation (CNCF) voted to onboard Hyperlight into their Sandbox program.

WASM - What's next?



composition of an
average code base

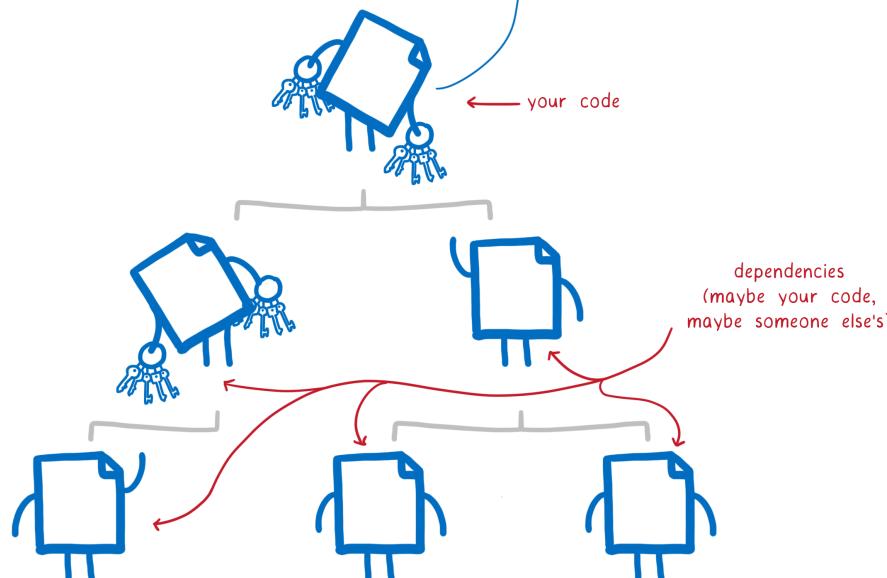


Dependencies

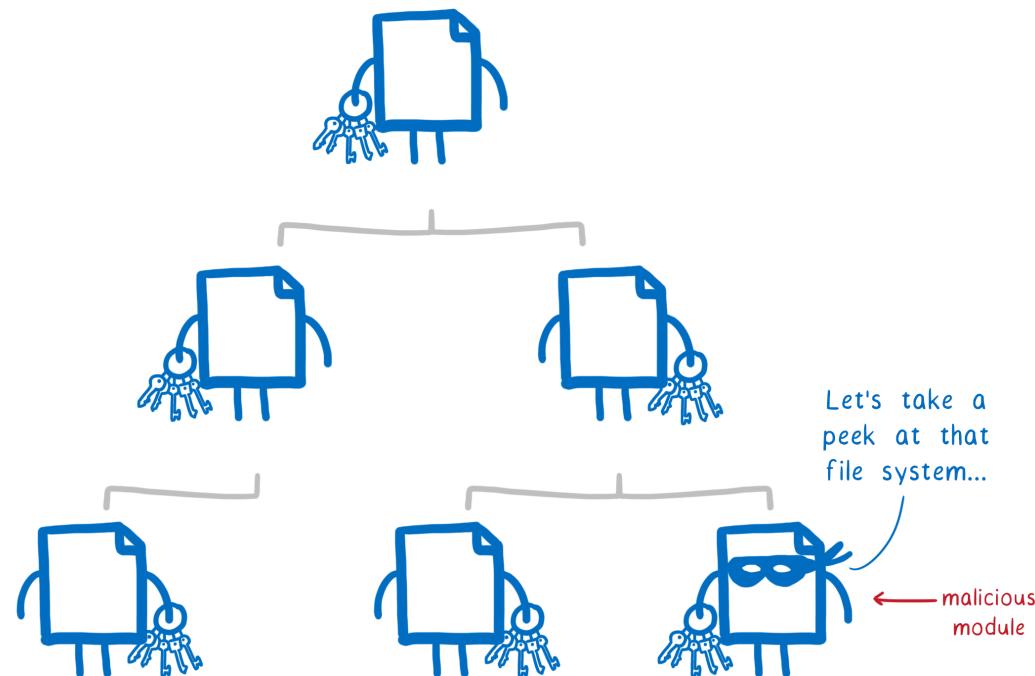


Here are the keys
to the castle.

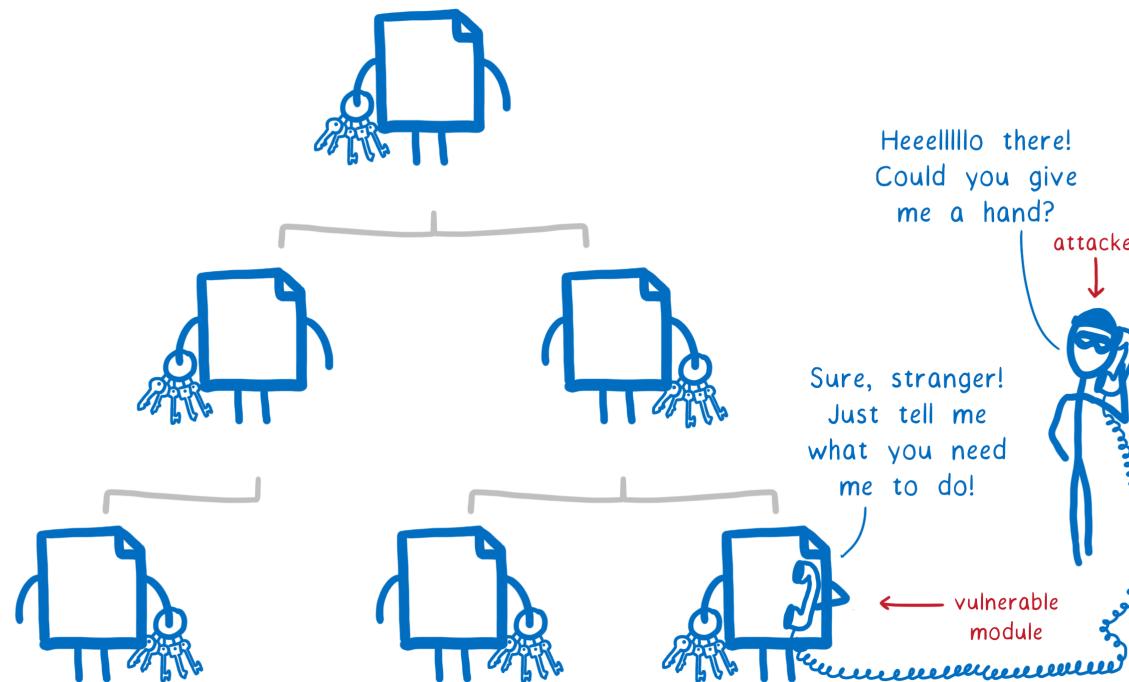
Make copies and
pass them down to
your dependencies.



Malicious module

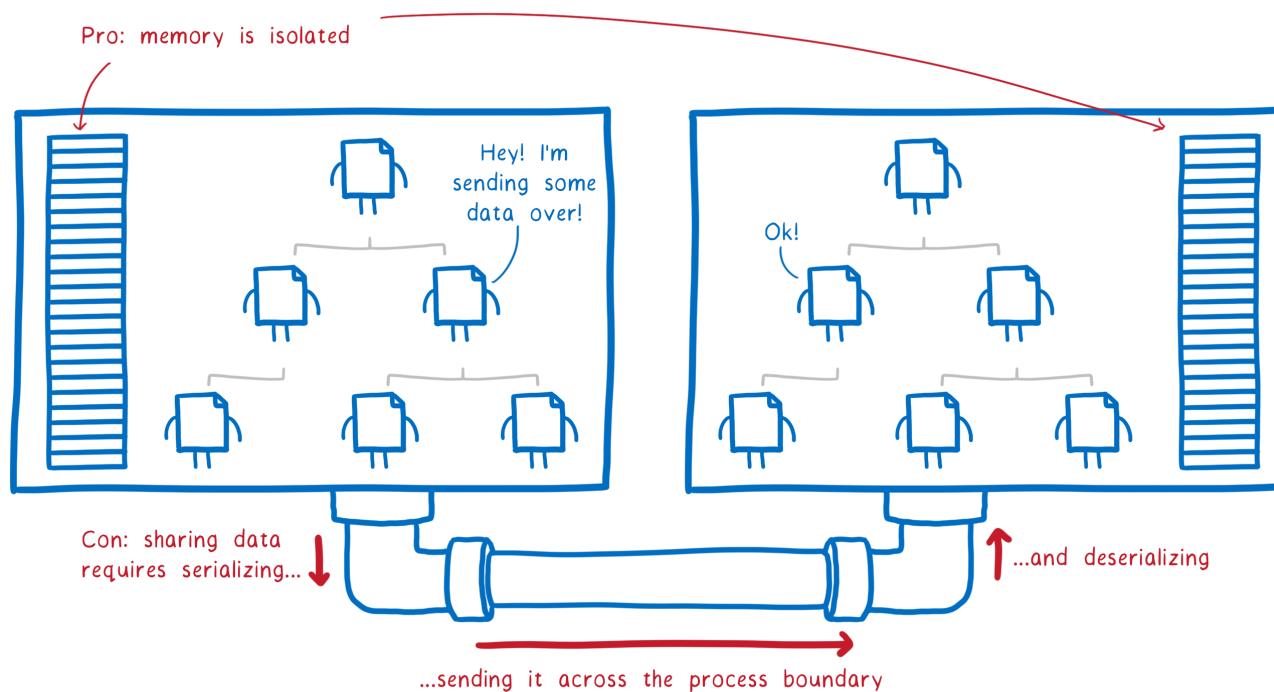


Vulnerable module



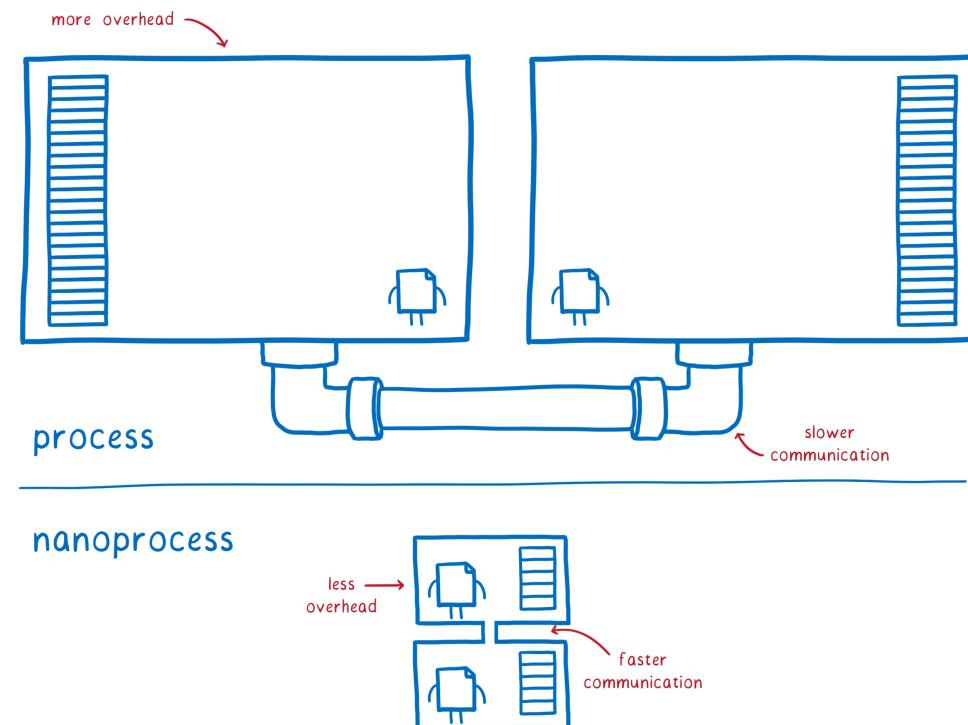


Process Isolation





WebAssembly Nano-Process



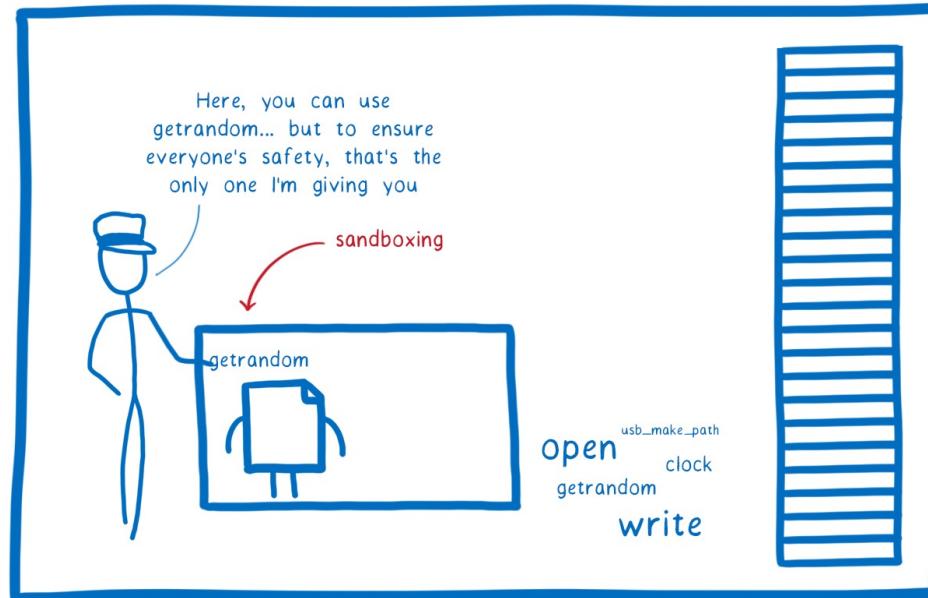
* not drawn to scale





WebAssembly Nano-Process

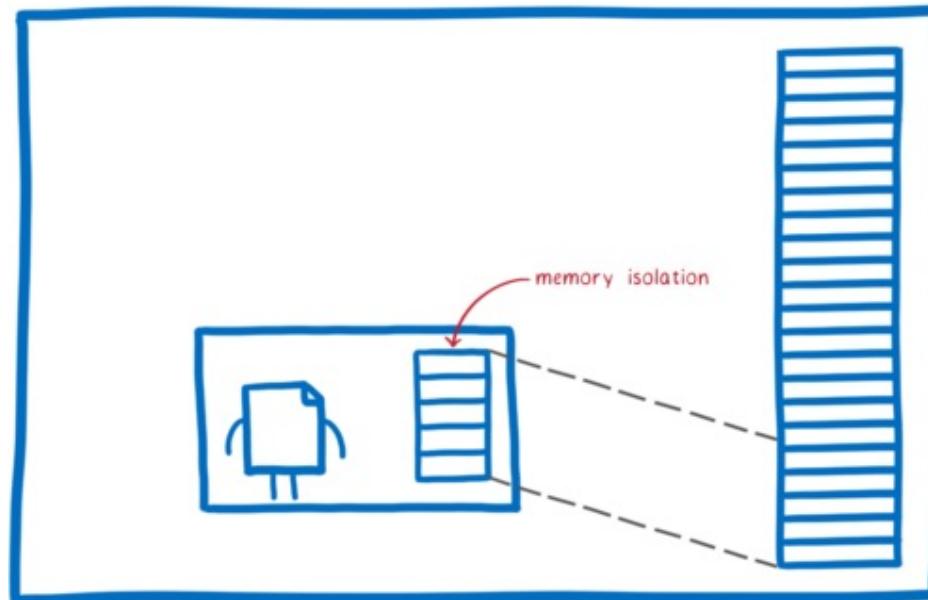
1. Sandboxing





WebAssembly Nano-Process

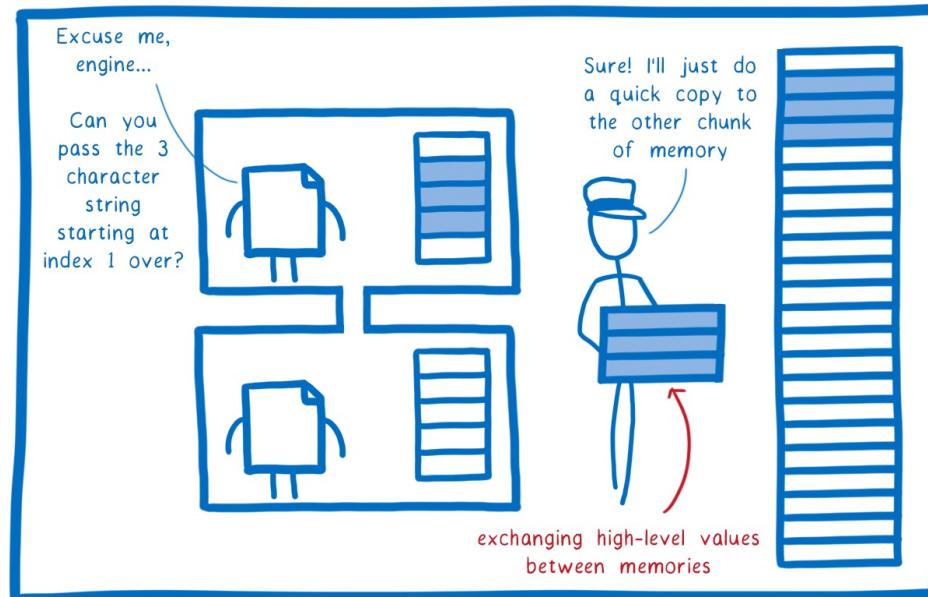
2. Memory model





WebAssembly Nano-Process

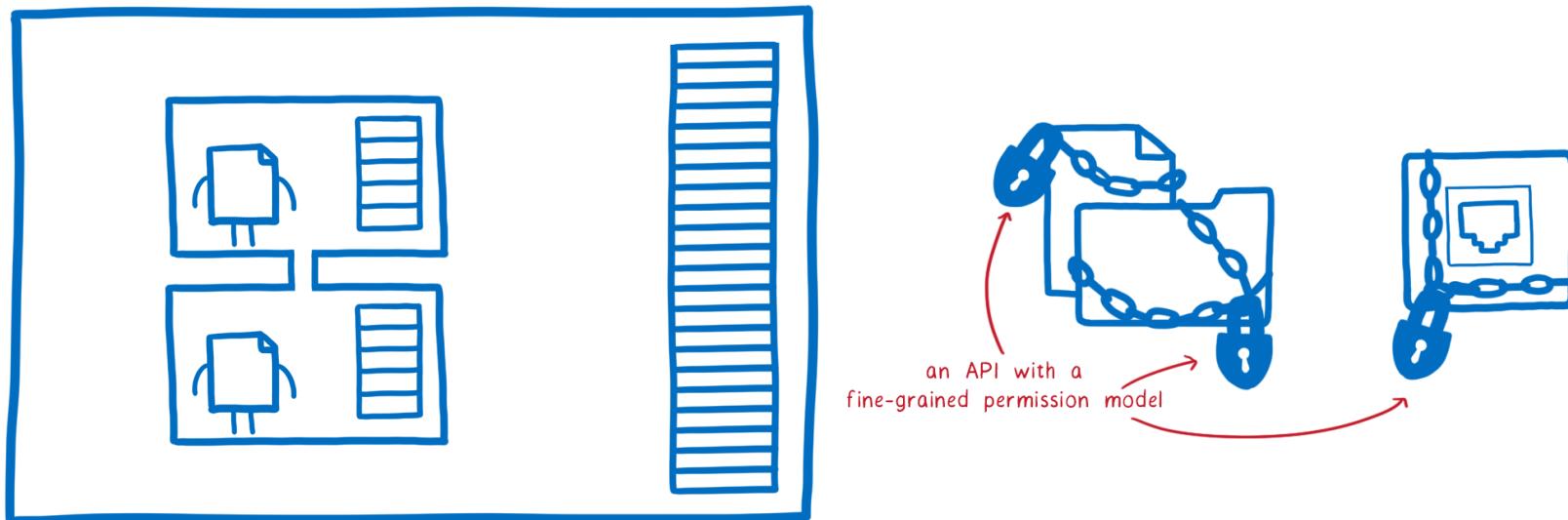
3. Interface Types





WebAssembly Nano-Process

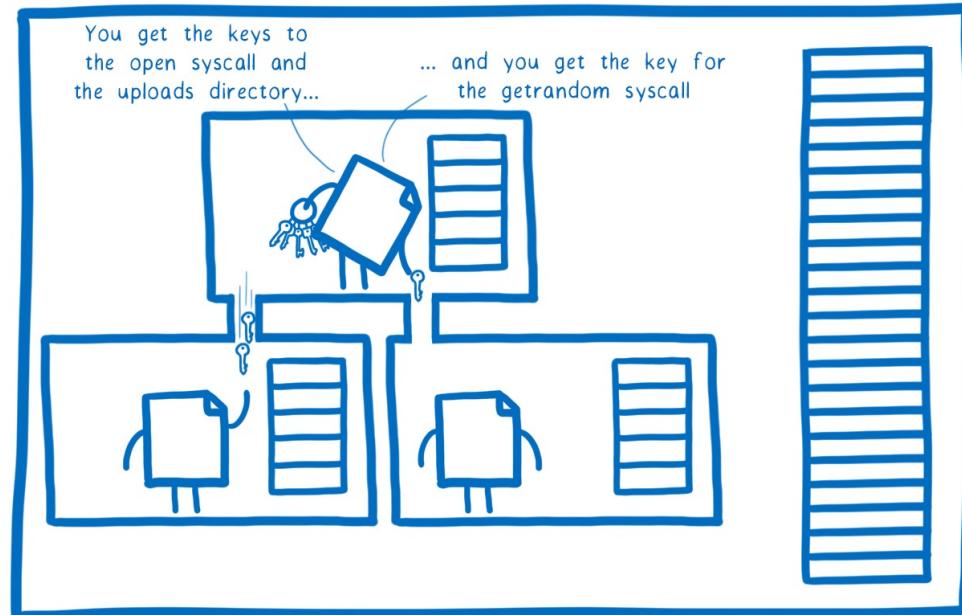
4. WebAssembly System Interface





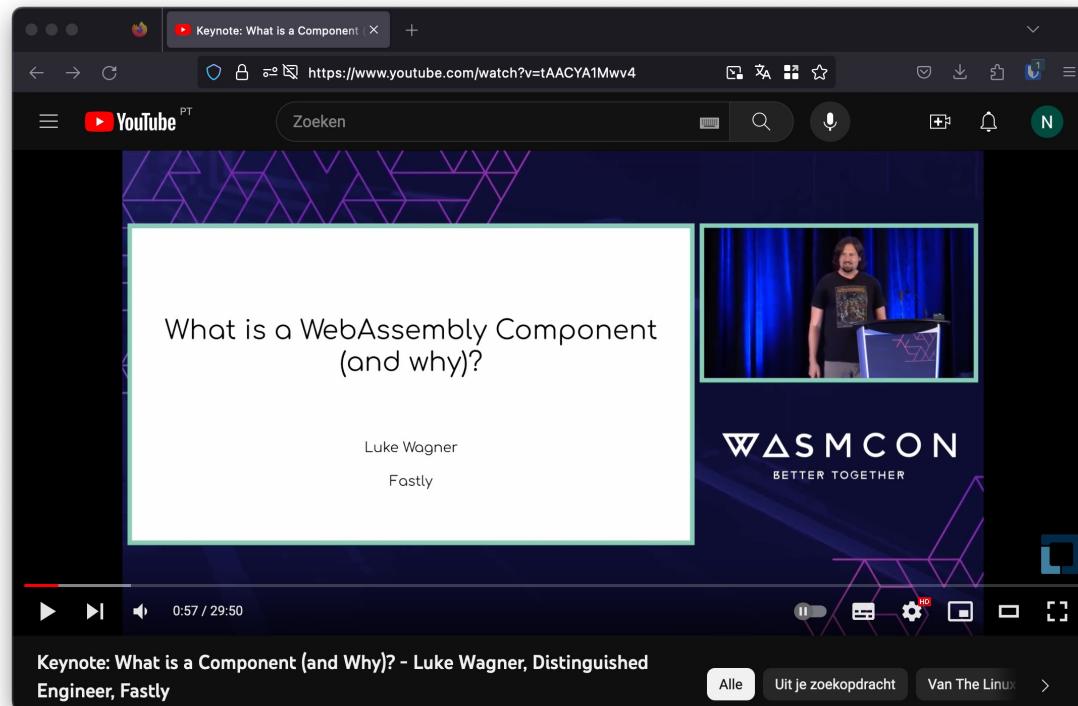
WebAssembly Nano-Process

5. The missing link





WebAssembly Component Model



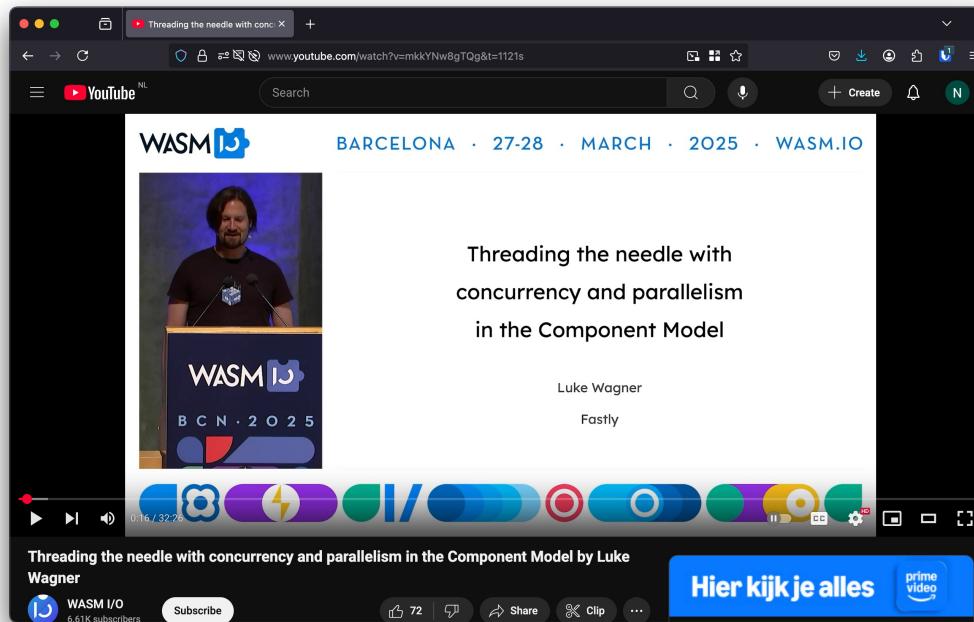
@niels.fennec.dev



@nielstanis@infosec.exchange



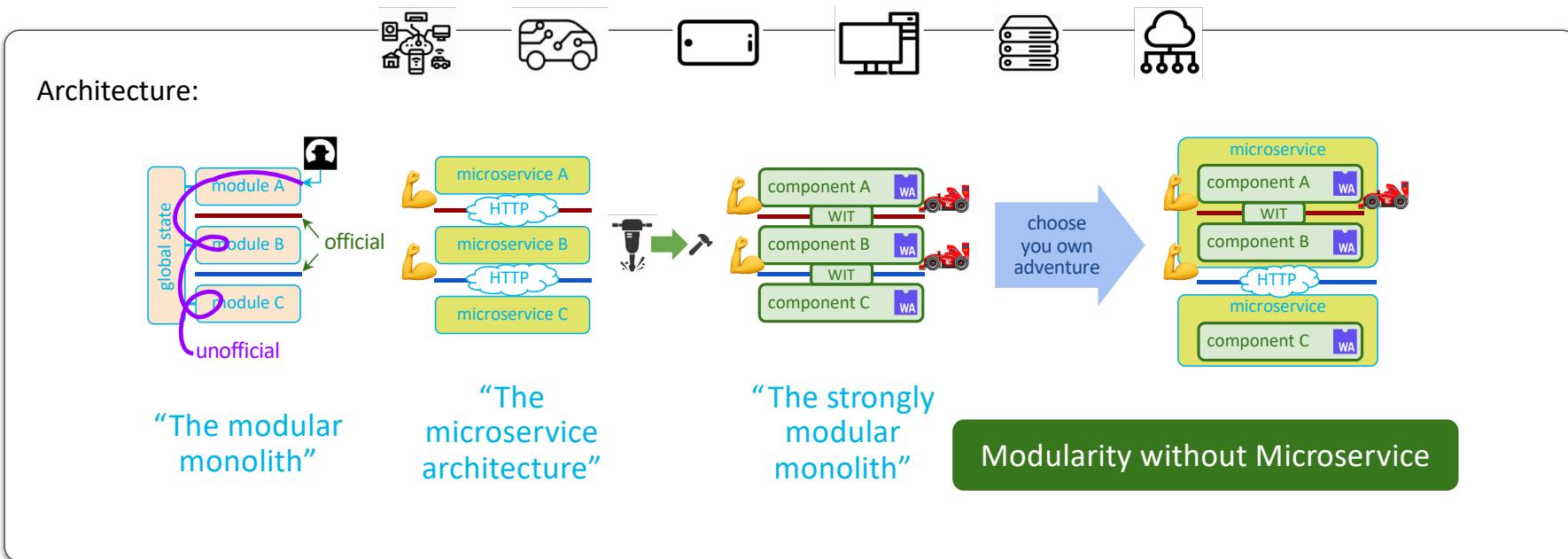
WebAssembly Component Model



 @niels.fennec.dev  @nielstanis@infosec.exchange

WASI 0.3

?





WASI Preview 2 (0.2)

A screenshot of a web browser window titled "WASI Preview 2 Launched - sunfishcode". The URL is https://blog.sunfishcode.online/wasi-preview2/. The page content is a blog post by "sunfishcode" titled "WASI Preview 2 Launched", posted on January 25, 2024. The post discusses the launch of WASI Preview 2 and its significance as a major milestone. It highlights the stability of the Preview 2 APIs, the cross-language and virtualizable nature of WASI, and the work involved in implementing the component model. The browser interface shows standard navigation and search controls.



🔗 @niels.fennec.dev 📩 @nielstanis@infosec.exchange



WASI 0.3

Roadmap - WASI.dev

wasi.dev/roadmap

WA SI WASI.dev GitHub

Introduction
Interfaces
Resources
Contribute
Roadmap

Previous releases:

| WASI version | Date |
|--------------|------------|
| 0.2.1 | 2024-08-01 |
| 0.2.2 | 2024-10-03 |
| 0.2.3 | 2024-12-05 |
| 0.2.4 | 2025-02-06 |
| 0.2.5 | 2025-04-03 |

The release train for WASI 0.2 will end following the release of WASI 0.3.0.

Upcoming WASI 0.3 releases

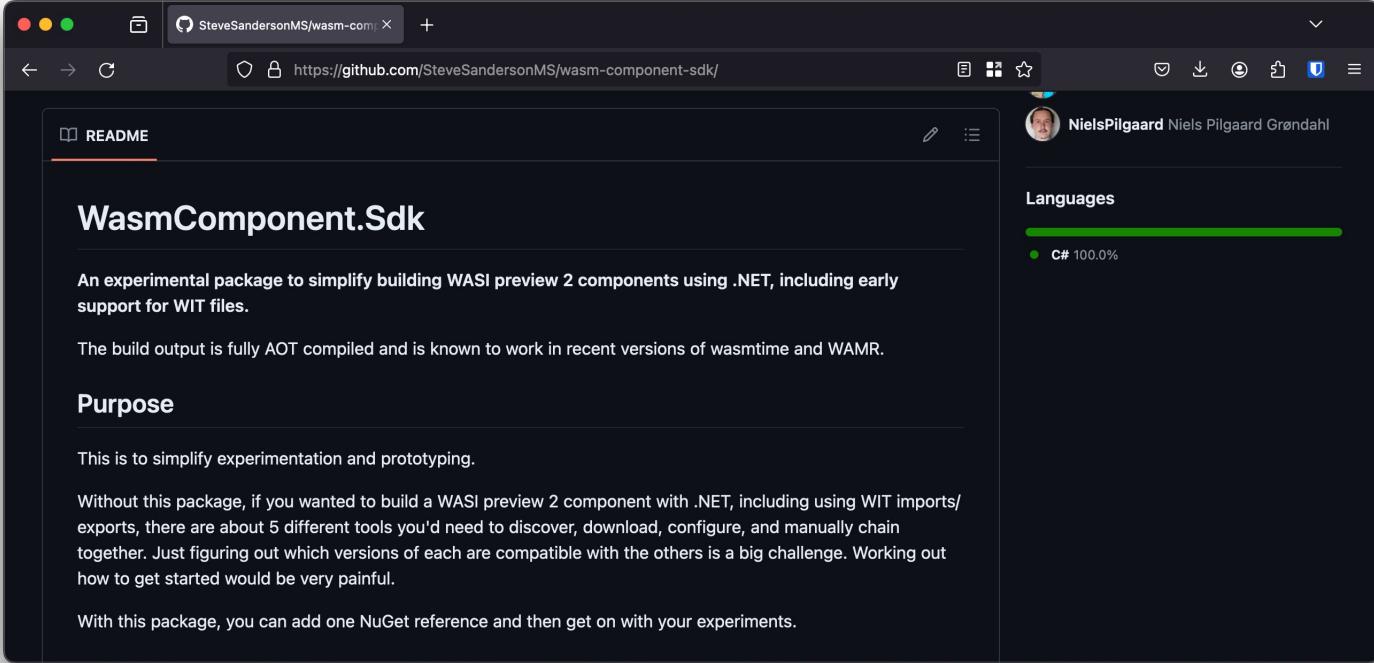
At this time, WASI 0.3.0 previews are expected in August 2025 and completion is expected around November 2025. Please see the [presentation](#) and [notes](#) from the May 2025 WASI SG meeting for more information.

WASI 0.3.0 will add **native async support** to the Component Model and refactor WASI 0.2 interfaces to take advantage of native async.





WasmComponent.SDK



The screenshot shows a GitHub repository page for `SteveSandersonMS/wasm-component-sdk`. The main content is the `README` file. It starts with a heading **WasmComponent.Sdk**, followed by a description: "An experimental package to simplify building WASI preview 2 components using .NET, including early support for WIT files." Below this, it states: "The build output is fully AOT compiled and is known to work in recent versions of wasmtime and WAMR." A section titled **Purpose** explains that the package simplifies experimentation and prototyping. It contrasts this with the difficulty of building components without the package, mentioning the need to discover, download, and manually chain together multiple tools. Finally, it notes that with the package, one can add a NuGet reference and begin experiments. On the right side of the screen, there is a sidebar for the repository owner, NielsPilgaard, showing their profile picture, name, and a languages section indicating 100% proficiency in C#.



 @niels.fennec.dev  @nielstanis@infosec.exchange



componentize-dotnet

The screenshot shows a GitHub repository page for 'componentize-dotnet' by 'bytecodealliance'. The page includes a README section, a 'Languages' chart showing 100% C#, and a sidebar with user profiles for 'ericgregory' and 'itowlson'.

README

componentize-dotnet

Simplifying C# wasm components

A [Bytecode Alliance](#) hosted project

If you have any questions or problems feel free to reach out on the [c# zulip chat](#).

Purpose

This is to simplify using Wasm components in c#.

Without this package, if you wanted to build a WASI 0.2 component with .NET, including using WIT imports/exports, there are several different tools you'd need to discover, download, configure, and manually chain together. Just figuring out which versions of each are compatible with the others is a big challenge. Working out how to get started would be very painful.

With this package, you can add one NuGet reference. The build output is fully AOT compiled and is known to work in recent versions of wasmtime and WAMR.

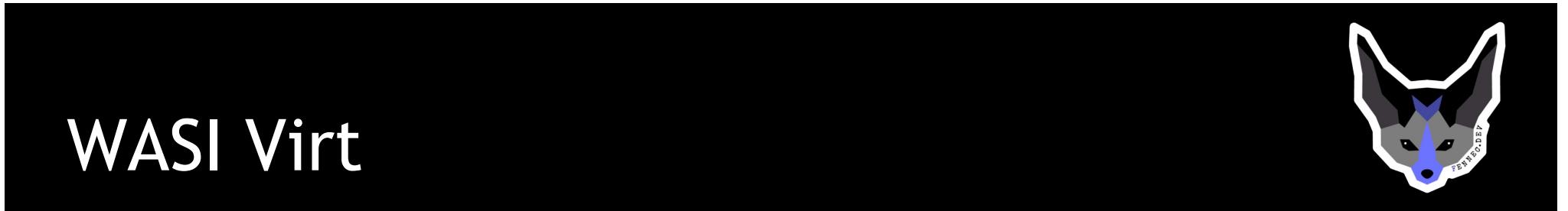
All the underlying technologies are under heavy development and are missing features. Please try to file it on the relevant underlying [tool](#) if relevant to that tool.

Getting started

Limitation: Although the resulting `.wasm` files can run on any OS, [the compiler itself is currently limited to Windows](#).



@niels.fennec.dev @nielstanis@infosec.exchange



WASI Virt

The screenshot shows a GitHub repository page for "bytecodealliance/WASI-Virt: Virt". The page title is "WASI Virt" and it describes the project as a "Virtualization Component Generator for WASI Preview 2" and a "Bytecode Alliance project". It includes a CI status badge showing "CI passing" and a "Languages" section indicating 99.5% Rust and 0.5% Shell. The main content area contains a README.md file with details about the component's composition, supported subsystems, and configuration options.

README.md

WASI Virt

Virtualization Component Generator for WASI Preview 2

A Bytecode Alliance project

CI passing

The virtualized component can be composed into a WASI Preview2 component with `wasm-tools compose`, providing fully-configurable WASI virtualization with host pass through or full encapsulation as needed.

Supports all of the current WASI subsystems:

- [Clocks](#): Allow / Deny
- [Environment](#): Set environment variables, configure host environment variable permissions
- [Exit](#): Allow / Deny
- [Filesystem](#): Mount a read-only filesystem, configure host filesystem preopen remappings or pass-through.
- [HTTP](#): Allow / Deny
- [Random](#): Allow / Deny
- [Sockets](#): Allow / Deny
- [Stdio](#): Allow / Deny / Ignore



@niels.fennec.dev @nielstanis@infosec.exchange

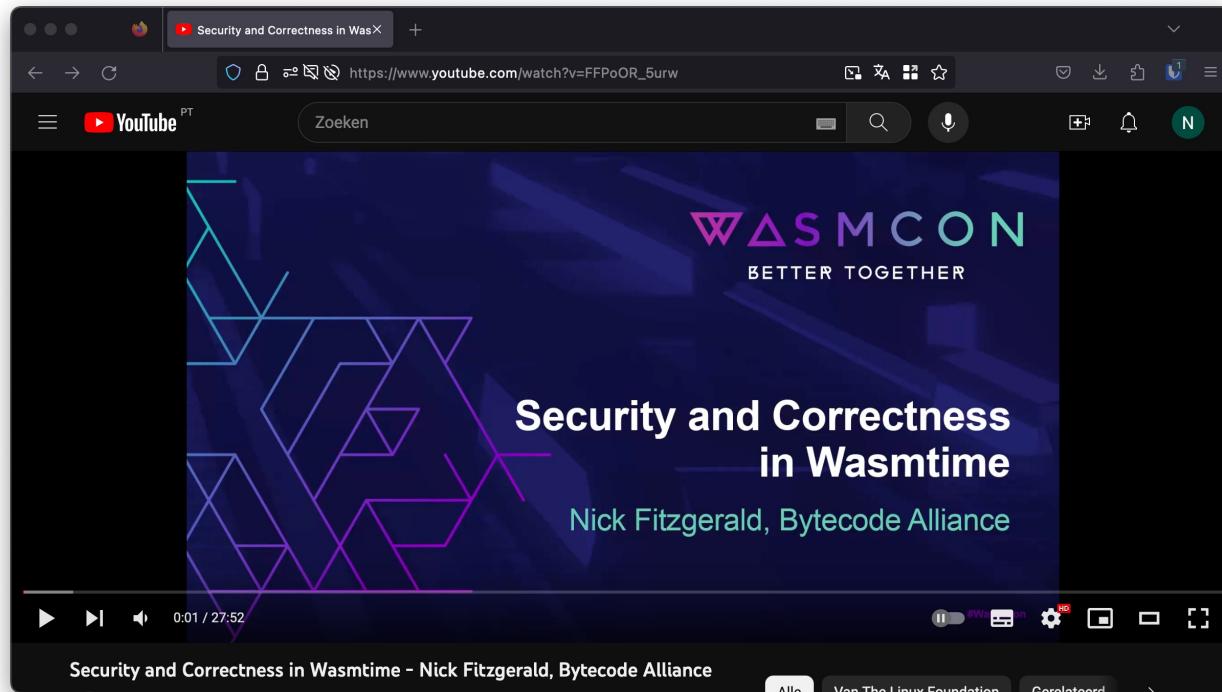


Runtimes and Security

- Most security research published focusses on correctness of WASM runtimes/VM's
- Bytecode Alliance Blogpost:
 - “Security and Correctness in Wasmtime”
 - Written in Rust → Using all it's LangSec features
 - Continues Fuzzing & formal verification
 - Security process & vulnerability disclosure



Runtimes and Security



@niels.fennec.dev



@nielstanis@infosec.exchange



WebAssembly Lineair Memory

Everything Old is New Again: Binary Security of WebAssembly

Authors:
Daniel Lehmann, *University of Stuttgart*; Johannes Kinder, *Bundeswehr University Munich*; Michael Pradel, *University of Stuttgart*

Abstract:
WebAssembly is an increasingly popular compilation target designed to run code in browsers and on other platforms safely and securely, by strictly separating code and data, enforcing types, and limiting indirect control flow. Still, vulnerabilities in memory-unsafe source languages can translate to vulnerabilities in WebAssembly binaries. In this paper, we analyze to what extent vulnerabilities are exploitable in WebAssembly binaries, and how this compares to native code. We find that many classic vulnerabilities which, due to common mitigations, are no longer exploitable in native binaries, are completely exposed in WebAssembly. Moreover, WebAssembly enables unique attacks, such as overwriting supposedly constant data or manipulating the heap using a stack overflow. We present a set of attack primitives that enable an attacker (i) to write arbitrary memory, (ii) to overwrite sensitive data, and (iii) to trigger unexpected behavior by diverting control flow or manipulating the host environment. We provide a set of vulnerable proof-of-concept applications along with complete end-to-end exploits, which cover three WebAssembly platforms. An empirical risk assessment on real-world binaries and SPEC CPU programs compiled to WebAssembly shows that our attack primitives are likely to be feasible in practice. Overall, our findings show a perhaps surprising lack of binary security in WebAssembly. We discuss potential protection mechanisms to mitigate the resulting risks.



@niels.fennec.dev @nielstanis@infosec.exchange



WebAssembly Lineair Memory

A screenshot of a PDF document titled "Securing Stack Smashing Protection in WebAssembly Applications" by Quentin Michaud, Yohan Pipereau, Olivier Levillain, and Dhouha Ayed. The document is viewed in a dark-themed browser window on arxiv.org. The abstract discusses the vulnerability of WebAssembly to buffer overflow due to lack of protection mechanisms and evaluates the implementation of Stack Smashing Protection (SSP) in standalone runtimes. The footer of the slide shows the date "23 Oct 2024".

Securing Stack Smashing Protection in
WebAssembly Applications

Quentin Michaud^{1,2}, Yohan Pipereau², Olivier Levillain², and Dhouha Ayed¹

¹ Thales Group, Palaiseau, France
firstname.lastname@thalesgroup.com

² SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France
firstname.lastname@telecom-sudparis.eu

Abstract. WebAssembly is an instruction set architecture and binary format standard, designed for secure execution by an interpreter. Previous work has shown that WebAssembly is vulnerable to buffer overflow due to the lack of effective protection mechanisms. In this paper, we evaluate the implementation of Stack Smashing Protection (SSP) in WebAssembly standalone runtimes, and uncover two weaknesses in their current implementation. The first one is the pos-

Conclusion



- WebAssembly has a lot of potential to be used to run, extend, and secure your applications!
- Its as secure as the WebAssembly runtime implementation!
- WASI 0.2 big milestone; tooling in progress!
- WASI 0.3 due in August 2025
- Cloud Native ❤️ WebAssembly

Merci! Bedankt! Thanks!



- <https://github.com/nielstanis/secappdev25wasm>
- ntanis at Veracode.com
- @nielstanis@infosec.exchange
- <https://blog.fennec.dev>