Research: WASM as a Lua alternative and for dynamically loadable modules

11/08/2019 09:18 PM - Jason Ish

Status: Assigned
Priority: Normal
Assignee: Jason Ish
Category: TBD
Target version: TBD
Effort:
Difficulty:

Description
This ticket is to capture thoughts and research about using WASM for:

- A Lua alternative: matching and outputs (where Lua is used)
- As a module format for dynamically loadable modules/plugins

WASM as a Lua Alternative

- WASM modules run in a very restrictive execution environment where they cannot access the network, files, etc.
- This might be OK for pure algorithmic uses, such as a module that calculates entropy.
- But is not suitable if the module requires access to external files, either to read in data from an external source (for example in a Lua rule), or writing to a custom log file (a Lua output).
- WASM also requires more overhead on the authors part. Once they have chosen a language, they will have to configure their toolchain to output WASM. While in some cases this may be trivial, it is more overhead than writing a Lua script.

WASM for Dynamically Loadable Modules

- WASM may be more interesting for dynamically loadable modules, but its restricted environment may not make that very popular.
- From my understanding it would not be possible to implement a custom database or Kafka style output as a WASM module. However I could be wrong as I've seen examples of Nginx recompiled to WASM, so more research is required here.
- Its restrictions may make it not popular as a format for dynamically loadable modules, however the strict environment it runs in would be nice. But writing native plugins and loading as a .so will ultimately be more flexible (but not sandboxed).

WASM outside of the browser also appears to be very young and rapidly evolving.

I also found AssemblyScript ([https://github.com/AssemblyScript/assemblyscript](https://github.com/AssemblyScript/assemblyscript)) interesting. This is a compiler for a subset of TypeScript that compiles to WASM.

Related issues:
Related to Task #3288: Suricon 2019 brainstorm
Related to Task #3307: Research: evaluate future of lua support in Suricata

History
#1 - 11/08/2019 09:18 PM - Jason Ish
- Description updated

#2 - 11/09/2019 08:50 AM - Victor Julien
When you say 'rust is less secure' this is because it is not sandboxed?

Do you have any sense of the runtime overhead of WASM?

#3 - 11/09/2019 08:50 AM - Victor Julien
- Related to Task #3288: Suricon 2019 brainstorm added

#4 - 11/09/2019 09:13 AM - Victor Julien
- Related to Task #3307: Research: evaluate future of lua support in Suricata added
Victor Julien wrote:

When you say 'rust is less secure' this is because it is not sandboxed?

Yes. I should have said a "native plugin" because its not sandboxed. Will update.

Do you have any sense of the runtime overhead of WASM?

No, I guess it would be good to define a workload and test between Rust, WASM, and Lua. I should be able to get a sense of the function call overhead pretty easily though.

- Description updated

For some very non-scientific benchmarking I created a function in each of Rust, WASM and Lua that simply took an i32 and returned that value incremented by 1. Then called this function 1,000,000 times. In the Rust case I summed the values returned to make sure the call didn't get optimized out. Though it may have got inlined (and its an unfair comparison).

For WASM and Lua, the function was loaded into memory once did reduce the loading of the module.

The WASM module was built with Rust in release mode.

Rust elapsed: 51ns  
WASM elapsed: 438.802712ms  
Lua elapsed: 157.969266ms

All this tells me is that the overhead of calling a WASM function is more than Lua though. I should probably do another test where some actual work is done in the loaded module.

btw - this may be a bit off topic - but it would be nice if there are similar profile counters in the rule profiling runs when Lua (or possibly different) scripts are used.(maybe a separate ticket)

To get an idea of performance I did iterations of sha256, one with a pure Lua implementation, and the other a pure Rust implementation compiled to WASM. One limitation is the Rust test runner I used did not make use of Luajit, instead plain Lua 5.3, so I also benchmarked Lua vs Luajit with a simple Lua script calling the same sha256 implementation.

I hashed the contents of the sha2.lua module 1000 times to come up with the following numbers:

Rust calling pure Rust: 635ms  
Rust calling WASM (build from Rust): 985ms  
Rust call Lua 5.3: 42s  
Lua 5.3: 42s  
Luajit: 750ms

The pure Rust/Rust is obviously the winner, there is no language boundary crossed. Lua (non-JIT) is slow. Luajit is very fast as well.

WASM is very close to Luajit, and it may be the performance difference is in the function calls crossing language boundaries.

I think its also worth noting that its non-trivial to pass non primitive number types into a WASM function. You first have to copy data into the linear WASM memory space. For example, in Rust you would pass a string into WASM like:

```rust
code
let host_string = "from Rust!";
// Write the string into the lineary memory
```
for (byte, cell) in host_string
    .bytes()
    .zip(memory.view()[0 as usize..(host_string.len()) as usize].iter())
    {
        cell.set(byte);
    }

    // Call our exported function!
    instance.call(
        "hello_string_from_rust",
        &[
            Value::I32(0),
            Value::I32(host_string.len() as _),
        ]);

I'm not clear at this time how you would do this if your WASM module also did some memory management.

#11 - 11/14/2019 03:16 PM - Pierre Chifflier

WASM is only a portable assembly, so in itself it will not be enough. Maybe WASI and/or the nanoprocesses, as described in [https://hacks.mozilla.org/2019/11/announcing-the-bytecode-alliance/](https://hacks.mozilla.org/2019/11/announcing-the-bytecode-alliance/) will help isolating applications while being able to define some communication API.