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
- Related to Task #4097: Suricon 2020 brainstorm
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.

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.

<table>
<thead>
<tr>
<th>Language</th>
<th>Elapsed Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust</td>
<td>51ns</td>
</tr>
<tr>
<td>WASM</td>
<td>438.802712ms</td>
</tr>
<tr>
<td>Lua</td>
<td>157.969266ms</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Elapsed Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust</td>
<td>635ms</td>
</tr>
<tr>
<td>Rust calling WASM (build from Rust):</td>
<td>985ms</td>
</tr>
<tr>
<td>Lua 5.3:</td>
<td>42s</td>
</tr>
<tr>
<td>Lua 5.3:</td>
<td>42s</td>
</tr>
<tr>
<td>Luajit:</td>
<td>750ms</td>
</tr>
</tbody>
</table>

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
define_text_string
    .param $1, i32
    .proc
    ;...
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/ will help isolating applications while being able to define some communication API.

#12 - 11/02/2020 07:30 PM - Victor Julien

- Related to Task #4097: Suricon 2020 brainstorm added


Some updates from the 2020 brainstorm:

- still an interesting approach
- sandbox/jail has cost: no zero copy of data
- "not a scripting language", so not a full replacement for lua in its current form
- lack of common libs and toolchain

#14 - 12/22/2020 01:07 PM - Pierre Chifflier

Hi,

I've continued working on this as an experiment on my free time, so here's an update.

I have a working branch of suricata, able to load output modules compiled to wasm. The C part is quite similar to lua: create an output module with similar configuration and C functions), and the WASM engine and code is written in Rust using `wasmtime`.

Basically, everything seems to be working, and I've been able to write a full TLS + X.509 parser in a wasm module using the rust crates, for example.

Since I feel it's important to have code and questions, I've uploaded some code to github, as well as documentation and test modules. To simplify, I've created a global repository linking everything as submodules here: https://github.com/chifflier/wasm-suricata

There are some basic instructions for setup (to be improved).

I've also used the wiki to write a more developer-oriented documentation: https://github.com/chifflier/wasm-suricata/wiki

#15 - 02/17/2021 05:55 PM - Jason Ish

Thanks for this work Pierre, it really shows that this is possible. Of course, it leads me to just a pile of questions.

Why an output module? What is the use case here?

Could the compilation and caching be done with a standalone app (or subcommand of Suricata). I'm think to reduce downtime during restart should a new module have been installed.

Given that multiple languages can target WASM, how realistic is it that we could support multiple languages? We'd probably want to provide some bindings to make some things easier, but we probably wouldn't have the capacity to provide these bindings in all languages. Does that make sense?

When I think of a use case for WASM plugins, I think the best/most popular use case might be an app-layer: the parser, detection keywords, and log formatting. Assuming we get to the point that we can do this with C/Rust, do you think this could be a good use case for WASM? I do worry that the copying back and forth of data could get expensive, but safety is something to consider as well.

I assume this wouldn't require WASI at all. All output would be done by Suricata-core.

Could such a parser, if written in Rust be easily migrated to a built-in parser rather than a WASM one?

I still don't see this as a realistic replacement for our Lua scripting. The primary users of that clearly seem to want a scripting language without all the hassles of a toolchain. Would you agree? With respect to this I mean scripts/modules used to by rules and shipped with rules. And then custom output scripting.
Thanks again.