Integrating Differential Privacy into Fluent Bit

Maintenance Manual

Julien Acker

April 26, 2024

Maintenance Manual

The use of the software package may be limited to UNIX-like operating systems as `set(FLB_FILTER_WASM No)` is present in `fluent-bit/cmake/windows-setup.cmake` therefore as of the current release you cannot run WASM filter plugins on the Windows-native version of Fluent Bit without changing that build flag (and thus running with an unsupported change). As the provided Dockerfile will do all Fluent Bit operations inside of a Debian Bookworm (Slim) container image, the use of Docker-on-Windows should be functional.

1 Using your host's Rust packages

If you wish to develop the application on your host, you will need to make sure a modern version of rust and cargo is installed on your system and available to your applicable PATH. In addition, the wasm32-wasi target must be installed if you wish to produce a WASM binary. This can be accomplished by either using Rustup or through your Linux distribution's native packaging.

- For Fedora and derivatives: `dnf install rust rust-std-static-wasm32-wasi cargo`
- For Debian and derivatives: `apt install rust-all libstd-rust-dev-wasm32` Note: Current Debian Stable release for libstd-rust-dev-wasm32 is 1.63.0 (similar goes for the rust-all package), all testing/development was done with the latest version of these packages which was packaged with Fedora and Rustup which both use the 1.77 series. Use Rustup if you wish to ensure compatibility.
- For any Unix-like system:
  - `curl -proto '=https' -tlsv1.2 -sSf https://sh.rustup.rs | sh`. Rustup is also packaged if you wish to avoid the security concerns that are inherently present when piping curl into sh.
  - Ensure that wasm32-wasi target is installed when prompted to specify your desired default target. Alternatively, `rustup target add wasm32-wasi` if rustup is already present on your host system.
- For Windows: download the upstream X86_64 Windows binary installer at `https://win.rustup.rs/x86_64`. Refer to Upstream documentation. This target OS was untested for this project, and thus, it is recommended to use (a containerized) Unix-like environment if possible.

However, none of the code is WASM-specific, so you can use the library compiled to any architecture which has support for all used dependencies. You are able to use the library from any programming language supporting the C Foreign Function Interface (thus, applicable C types). Note: wasm32-unknown-unknown is an unsupported target. We require the WASI environment to provide an interface for file I/O operations, in addition, rand requires manual intervention to specify the source of randomness where none can be assumed. The application is designed to panic if an improperly formatted UTF-8 tag variable or JSON-formatted records variable is passed to the filter plugin. User input nor log corruption should not be able to invoke a panic as Fluent Bit internally sanitizes inputs. Fluent Bit is the one responsible for correctly formatting strings / encoding JSON so shouldn't be possible in normal operation. The only time in development that this was triggered was when passing 0 length values into either case. It was deliberately decided that panicking in these cases is acceptable.

1.1 Build manually

Once you have Rust and Cargo installed with the `wasm32-wasi`, you can build the WASM binary with:

```
cargo build -release -target wasm32-wasi
```

This binary cannot be directly run and needs to be loaded by an external program (ie Fluent Bit).

Do note that if you have rust-analyzer installed, you can develop the codebase with confidence that if Rust Analyzer doesn't detect any problems, then the application likely functions without any further intervention. The issues I've had that weren't captured by Rust Analyzer were mostly "functional bugs" as in, was already captured by error handling (at the time `.unwrap()` however better practice regarding panicking have since been adopted) or weren't wholly Rust problems (the most notable is that formally I didn't convert the rust string to a CString, like Fluent Bit's own Rust WASM example, which sometimes worked but sometimes deallocated as this memory should be `free()`ed by Fluent Bit itself but I encounter a situation where Rust would sometimes free the string and sometimes not so some events were dropped and some weren't. Well they were all deallocated by Rust. But the pointer was handed over to C was for that string's former location). Therefore when it did work, it was due to unintentional use-after-free from Fluent Bit of the Rust string. You need to put the string into a CString so that Rust doesn't deallocate it when it passes the Foreign Function Interface barrier.

2 Apple Silicon Mac patch

In order to build on an Apple Silicon Mac, the following patch may be applied to the fluent-bit submodule. This is needed at least when running on Asahi Linux while using WAMR's AOT compile option. WAMR autodetects aarch64 when optimizing for the current architecture, however, when Fluent Bit attempts to load the binary it complains that it was expecting "aarch64v8" and received "aarch64". Going by the documentation, intended behaviour is that anything built for aarch64 should run on any aarch64 varient, however this may be a Fluent Bit configuration error.

```
frame=lines,
framesep=2mm,
baselinestretch=1.2,
fontsize=\footnotesize,
linenos,
breaklines, breakafter=/.-]{c}
diff --git a/lib/wasm-micro-runtime-WAMR-1.3.0/core/iwasm/aot/arch/aot_reloc_aarch64.c b/lib/wasm-micro-runtime-WAMR-1.3.0/core/iwasm/aot/arch/aot_reloc_aarch64.cindex b4bb6024a..4593cd706 100644
--- a/lib/wasm-micro-runtime-WAMR-1.3.0/core/iwasm/aot/arch/aot_reloc_aarch64.c
+++ b/lib/wasm-micro-runtime-WAMR-1.3.0/core/iwasm/aot/arch/aot_reloc_aarch64.c
@@ -56,7 +56,7 @@ get_target_symbol_map(uint32 *sym_num)
 #if (defined(__APPLE__) || defined(__MACH__)) && defined(__arm64__)
 #define BUILD_TARGET_AARCH64_DEFAULT "arm64"
 #else
-#define BUILD_TARGET_AARCH64_DEFAULT "aarch64v8"
+#define BUILD_TARGET_AARCH64_DEFAULT "aarch64"
 #endif

 void
```

Code Listings 1: Apple Silicon patch

This patch will need to be written to a file, `COPY` over to `fluent` stage, and applied to the `/src/fluent-bit` directory with the `git apply <filename>` command. Ensure filebreaks are correctly copied over.