# LLVM and the state of sanitizers on BSD

**David Carlier**

- French software engineer living in Ireland.
- Contribute to various opensource projects directly or indirectly related to FreeBSD and OpenBSD mainly, from enterprise solutions to more entertaining ones like video games.
- Contributor LLVM since end of 2017, committer since May 2018.
- Used to write for BSDMag.

**Status on FreeBSD and OpenBSD**

How it had started ?

- It often starts comes with initial frustration.
- Indeed, after having tried fuzzer under Linux and quite pleased with the outcomes, I realized it was not supported under FreeBSD.
- From this point, you can try to solve this “issue” by contributing.
- After a certain time, patience and relentlessness, they might give you the commit accesses.

**What is implemented in FreeBSD ?**

- address sanitizer (asan).
- thread sanitizer (tsan).
- memory sanitizer for the x86/64 architecture (msan).
- cache frag part of the efficiency sanitizer (esan).
- undefined behavior sanitizer (ubsan).

*Also the following components*

- libFuzzer.
- X-Ray instrumentation.
- Safestack.
- libCFI

*What could be ported*

- the working sets part of esan.

*Difficult to port*

- lsan due to lack of the stop the world feature in the kernel.

**What is implemented in OpenBSD ?**

- ubsan
- libFuzzer
- X-Ray instrumentation.

*What won't be ported*

- other sanitizers “due” to ASLR not possible to disable it also not possible to map large regions of memory for the shadow mappings.
- Safestack, even if it was possible, has not much of interest as OpenBSD has similar features out of the box.

**What are the key roles of the sanitizers ?**

They allow to detect at runtime some well know bugs especially the ones which are difficult to detect in production environments.

- msan is mainly all about detecting not initialized pointers.
- asan is more for memory handling as double and use after free ; heap and stack overflows ... minus the memory leaks detection under BSD as leak sanitizer is not doable at the moment. asan is mutually exclusive with msan.
- ubsan for errors about integer overflows (typical cases are the shift operations), misaligned pointers due to casts with different alignements.
- tsan to detect race conditions in multithread contexts.
- esan basically helps to pack the data structures efficiently to avoid cache fragmentation, as you would try to do manually with tools like phole for example.

**What is fuzzing all about ?**

- It is a testing technique, “invented” in late 80's by Barton Miller, when basically you try to give random data to your software and its dependencies included.

- Inputs source come from what call “corpus”.

- It is good to find particular set of bugs, based on input handling basically while trying to cover as much as possible code paths by mutating these inputs.

- Ideally running long, as the data will undergo some “mutation” in the process,  as necessary until it crashes eventually.

- Completes traditional unit tests set too.

**What corpus means ?**

- Place holder for inputs, inputs which suit the particular software.

- Let's imagine an image library reader which relies on specific binary format header to recognize if it is png/jpeg and so on.

- The corpus will then contain hand crafted data for a particular test.

- There is possibly more than one corpus but those corpus could possibly be merged, keeping only the relevant mutation results.

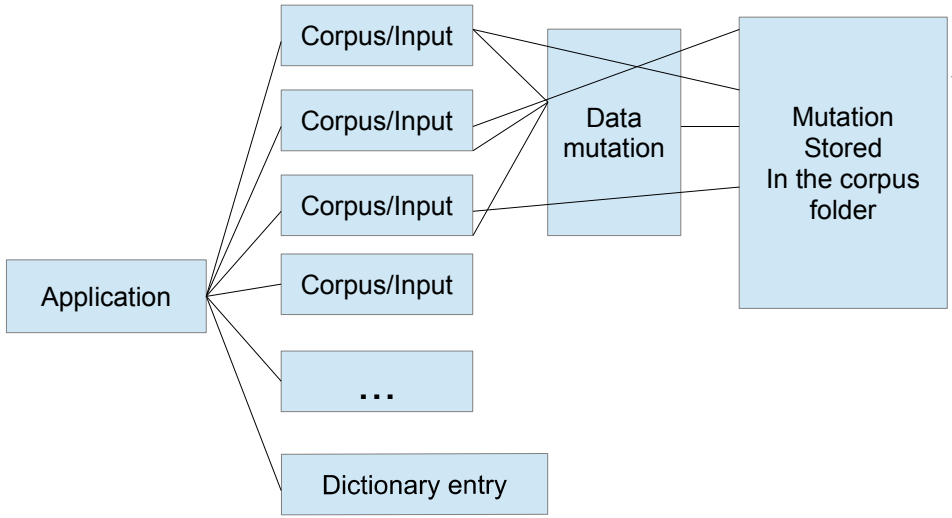
- Those mutations will be then stored (and to be reused) in this corpus.

**What mutation means in libFuzzer context ?**

- Mutation simply means some bytes are deleted, some others are inserted, got shuffled at random offsets. A dictionary (format key=value) can be used too.

- Software developers might start to have concerns ... That would trigger a segfault, stack overflows or SIGBUS throwing.

**Fuzzer workflow**




**How does it works under LLVM ?**

- Basic flag is -fsanitize=fuzzer to gives to either the C or C++ frontend.
- The code in question needs to have a specific entry point at minimum to receive the input data, main is already present. Saying that, there is an optional startup function to implement which catch the arguments.
- You can also implement the mutation and combination part.
- Can be combined with another sanitizer flag as ubsan, asan, msan, even lsan ...
- Once the binary built, has plethora options as parallel jobs, enabling/disabling signal interceptions, memory usage limit, ... some options are dependent to other sanitizers for example detecting memory leaks.

**We said options ?**

- A fuzzed library run each time the whole process, we can limit the number of runs. Limits the max length of the incoming inputs.
- Can be ran with parallel jobs.
- Enable/disable certain signal interruption.
- Limit the memory usage.
- Control the degree of mutation.

**Culprits of the fuzzing ...**

- There are certain type of softwares more difficult to fuzz.
- For instance, softwares listening on a socket !
- Example with h2o which is a web server library and uses libFuzzer for its tests.
- Indeed, in this case there is a need to create a whole handcrafted configuration, creating a client request from this fuzzed data. Fortunately, h2o is mainly a shared library, things get complicated if it s a monolithic binary.

**Xray Instrumentation**

Is a run time call tracing facility. Mainly made for function timing measurements.

Can be refined by explicitly tracing or not tracing certain functions via clang attributes, configuration files or at least by function thresholds.

Can be enabled/disabled at runtime.

When disabled, the performance overhead is usually non existent but has a more noticeable performance difference when enabled but somehow suited to be ran in production.

But usually only to be run for a certain time and for a subset of functions in order to collect enough data dependent also on the function threshold and the memory usage limit wish for the in memory buffer data collection.

**How does it work ?**

- Xray injects instrumentation hooks at function entry and exits.
- Empty hooks until xray is enabled so as runtime thus replaced by cycle counter, function identifier, thread id, base address metadata.
- The basic flag is -fxray-instrument
- Our binaries contains now xray_instr_map and xray_fn_idx sections within data segments.
- We need means to extract those data from the Elf binary to generate our call graphs => llvm-xray.
- Accounting is also a feature to display where the code spends most of the time.
- There is logging options settable via XRAY_OPTIONS.

**Accounting is not what you think !**

- But Is more about to display the code used and the cumulative time spent for each.
- If it is a multithread program, data can possibly be aggregated.
- Can be sorted by any column, formatted as csv
- Can give a good idea of the possible bottlenecks.

**Xray workflow**