CSE 231 Project Proposal

Breaking Apart the Shared Object Monolith

Farid Zakaria <fmzakari@ucsc.edu>
Achilles Benetopoulos <abenetop@ucsc.edu>
Kai Tamkun <heimskr@ucsc.edu>
Mitchell Elliott <miclelli@ucsc.edu>

Motivation

The ELF object file format has dominated the Linux space as the de-facto format in which all executables and shared object files are created. It is a highly configurable specification that has over time become more dependent on shared conventions such as the way in which dependencies are encoded.

Although the object format has helped solve many of the problems encountered in our heterogeneous world such as varying ISA, ABI and byte encoding, it can be difficult to leverage due to it being a concise packed binary-encoded format all placed within a single file.

Recent technology ideas have ushered a new software development and deployment model based on the ideas from the Nix thesis [1]. The central idea of the new "store model" is the placement of all necessary files (executables, configuration, shared objects) within a fixed read-only/immutable root.

Given this new software deployment model, is it necessary to encode all binary data within a single file? We would like to explore the idea of separating some of the data encoded within the ELF format into separate files so that they can more easily be modified and introspected.

Artifact

Phase 0

As a starting off point, we plan to manually decompose a few select example binaries into individual files (i.e. string table into a simple textual file) to demonstrate what an "exploded" ELF binary may resemble, and to motivate the rest of the work, while also highlighting further research work into a linker or kernel module which can understand this new multi-file executable format. For subsequent phases, we will focus on shared library symbols.

Phase 1

We will build a library that leverages gLibc's auditing API [1] for the dynamic linker to extract symbol resolution data and encode it into a separate file; likely a sqlite database. This file will contain the result of the symbol, offset and the shared object library that the symbol will have resolved to. Since symbols are typically resolved lazily, we can force the linker to do this immediately using the LD_BIND_NOW environment variable.

Phase 2

This phase is less clear what we can achieve, however we would like to explore the idea of changing the values in the sqlite database and have the dynamic linker load the modifications rather than what exists in the ELF header, as an alternative to something like LD_PRELOAD.

Evaluation

Much of the evaluation may be subjective, however we plan to document and write about the experience of having some of the packed ELF header.

Using the sqlite database for a view program, we can try to draw some interesting conclusions from running queries against it or perhaps extrapolate what could be done in the space of tooling as potential avenues for future work (e.g. symbol servers).

Resources

Laptop development should suffice for hardware resources. Knowledge about linkers and process execution is difficult to come by as little has been formally written on the subject (i.e. no textbooks). Any introductions or mentors who can help answer or guide some of our questions would be much appreciated.

Related Work

Some of the research ideas here are follow-up work by Farid's research paper "Mapping out the HPC Dependency Chaos" which he will present at SuperComputing 2022. Furthermore, there is ongoing research work into what a new compiler/linker and OS runtime could look like with the research direction Google's Fuchsia kernel is exploring.