CSE 231 Project Midterm Update

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>
https://github.com/fzakaria/CSE231-Breaking-Shared-Object-Monolith

Overall Project Status

Goal	Status
Phase 0: Create Various ELF Decompositions	Not Started
Phase 1: Create utility to record dynamic symbol bindings	Completed
Phase 1 Continued: Come up with interesting SQL analysis	In Progress
Phase 2: Use symbol database to replace LD_PRELOAD	Won't Complete

Update and Response to Feedback

Phase 1

A lot of hours were spent completing Phase 1 to the point where the pain of developing further validates the authors view point that the format is challenging to work with. The authors had to ramp up on C/C++, the ELF specification, dynamic symbol resolution and parts of ELF that are not even well specified (i.e. GNU DT HASH).

Phase 2

It was incredibly challenging to craft the C/C++ code [ref] necessary to record the dynamic symbols. It's not even clear yet if it's technically feasible to complete Phase 2 but it's looking like a solution will not be complete.

Instead we choose to come up with more interesting usages and SQL analysis of the symbols database to demonstrate an interesting conclusion.

Why it is helpful to split an ELF into multiple elements

While ELF has proved its usefulness as a binary format, it is not really conducive to introspective querying or lightweight modification, owing to its tight encoding and the inherent complexity associated with the format.

For instance, in order to determine whether a given shared library has a given symbol, the ELF header encodes not only all symbol names but a hash map and bloom filter for efficient lookup. Asking various questions about the nature of symbols (i.e. which symbols come from library X?) requires low level C/C++ coding or stitching together a hodge-podge of tooling.

Through completion of Phase 1 it's subjectively clear that reading and working with the current ELF format is challenging. It's incredibly terse, compact and at times weakly specified or hardly at all. A declarative interface such as SQL has already proved subjectively a very intuitive way in which to introspect.

Splitting an ELF file up into multiple components allows us to represent each section of interest into an appropriate format that facilitates introspection, investigation and modification.

Similarly to the symbol section, we believe that other sections of the ELF header will also benefit from such treatment. Our Phase 0 approach will demonstrate the idea of having the *String Table* in separate UTF-8 newline delimited files. Auditing the strings used by a particular binary can now easily leverage Unix tooling (i.e. sed), and can be modified – since the ELF format is a tight binary encoding, changing a string to be larger requires zeroing out the previous one and fixing up all references to the previous to point to the new larger section which is heavy handed.

Original Proposal

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.