Segregated Dynamic Linking with ELF

by Vivek Das Mohapatra, August 10, 2018

Introduction

In this article I'm going to discuss a project I've been working on at my employer (Collabora Ltd) for one of our customers (Valve Software): Segregated Dynamic Linking for ELF binaries.

Contents

Terminology

calling convention

The rules explaining how arguments and return value(s) should be passed between calling code and called function.

DSO

Dynamic Shared Object: A program, a library or a loadable module used to make a working dynamically linked application.

ELF

Executable and Linkable Format: A specific DSO format used by many unix-ish platforms.

foreign function

A function invoked by one DSO that is defined in another.

GOT

A table of Relocation Records (RRs), one for each external symbol.

linker

Specifically, the run-time dynamic linker - the part of system responsible for taking a set of DSOs and assembling them into a working, running program.

PLT

Procedure Linkage Table.

RR

Relocation Record.

stack

Area of memory where arguments for [and return values from] functions are stored during function calls.

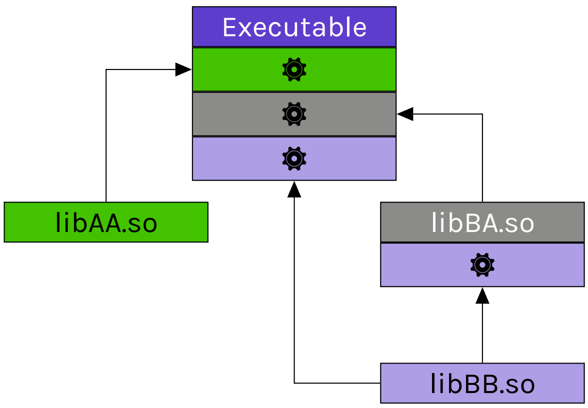
symbol

A function or variable (a named thing) provided by or used by a DSO.

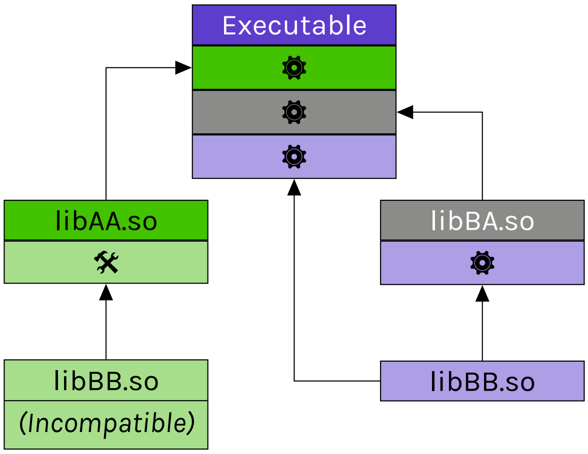
What is Segregated Dynamic Linking?

In normal dynamic linking there is one copy of every DSO mapped into memory, and each DSO can see the symbols of every DSO *that it asked to see.*

Example: If my program needs two DSOs (libAA and libBA) and both the program and libBA need a third DSO (libBB) then there will be one copy of libBB and both the main DSO and libBA will see libBB's symbols, but libAA will not.



In Segregated Dynamic Linking, we still link in a DSO as usual but we hide some of the symbols (functions, variables, etc) that would normally be visible as a result: This allows us to limit the visibility of symbols that would otherwise cause conflicts.



Why is Segregated Dynamic Linking?

It would be perfectly reasonable, at this point, to ask why we don't 'just' update the offending library to use a compatible version of its dependency:

In an ideal world that is exactly what we'd do - unfortunately that is not the world in which we live.

In this particular case the scratch that needs itching is the ability of a game in a runtime (to oversimplify: a fixed set of libraries guaranteed to be at specific versions) to work with a Mesa (3D graphics library) pulled from outside the runtime.

Why is this necessary? It falls out like this:

The runtime is effectively a promise made to the game developers that certain versions of a given set of libraries will always be available, no matter which version of the host operating system the game finds itself on.

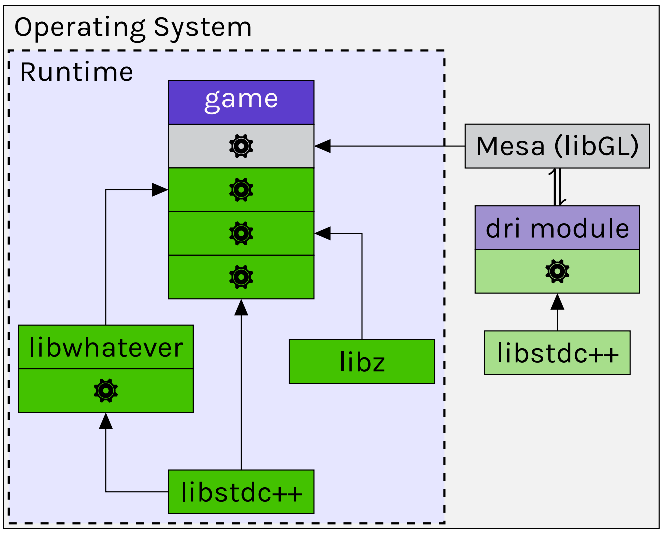
But Mesa needs to talk to specific hardware - and hardware changes rapidly. So we can't drop a frozen-in-time version of Mesa into the runtime and use that: Even though the Mesa interface might not change the old Mesa probably wouldn't be able to talk to shiny new hardware - which would mean your game simply wouldn't run on hardware made any significant amount of time after its release.

Ok, you might reasonably ask, why not 'just' pull in the Mesa from outside the runtime and be done with it? It turns out this is where our troubles begin:

- Mesa (or rather its hardware drivers) depend on `libstdc++`
- The runtime provides a specified `libstdc++` version
- Our game loads the runtime `libstdc++` DSO
- Our game loads the Mesa DSOs
- The linker notes that it already has a `libstdc++` DSO
- The linker uses this DSO to link the Mesa DSOs
- If the versions aren't compatible… we crash 💥

This is not, as you can imagine, a great sales pitch for your runtime: You can work around it by carefully controlling both the host operating system *and* the runtime - but that limits your portability.

That, then, is the initial problem. Our solution will look something like this:



How do we achieve segregated linking?

To answer that question we're going to need to pay attention to some low-level details of how function calls between DSOs work:

Calling foreign functions from a DSO

When code in a DSO wishes to call a foreign function it cannot do so directly: It has no way of knowing where in memory the linker has put the other DSO. This is fixed (as with all problems) by adding a layer of indirection (actually two, because it makes things tidier). Here's how it works:

Some definitions:

PLT - Procedure Linkage Table

An array of stub functions, one for each foreign function.

The Procedure Linkage Table (PLT) resides in the Read-Only TEXT section of the DSO, along with the rest of the executable code, and is shared by all copies of the DSO in use.

Every DSO has its own PLT.

GOT - Global Offset Table

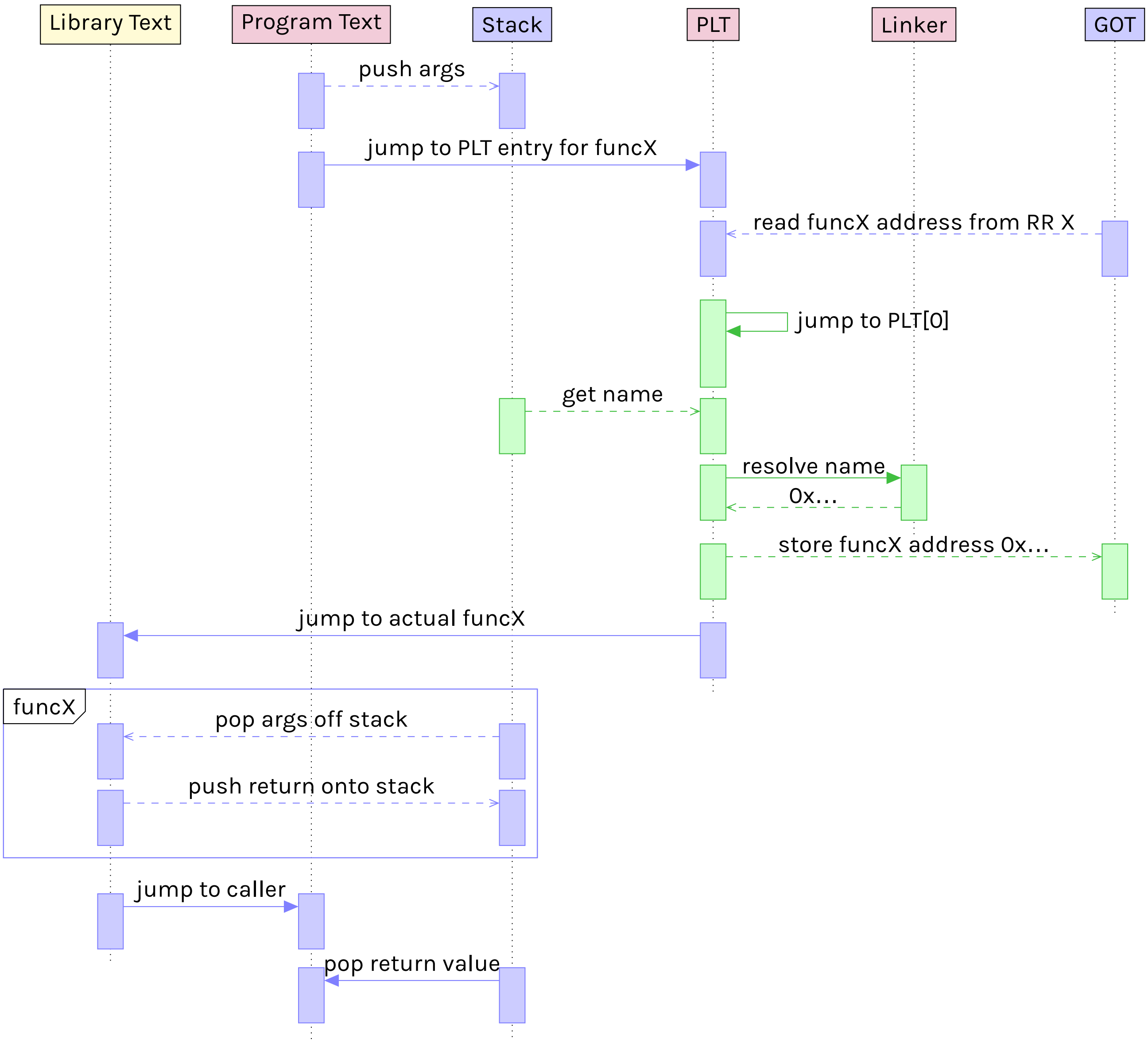
An array of RRs, each giving the actual location of an item mentioned in the PLT.

The GOT resides in the read-write DATA section of the DSO, there is a private copy for each copy of the DSO in use.

Every DSO has its own GOT.

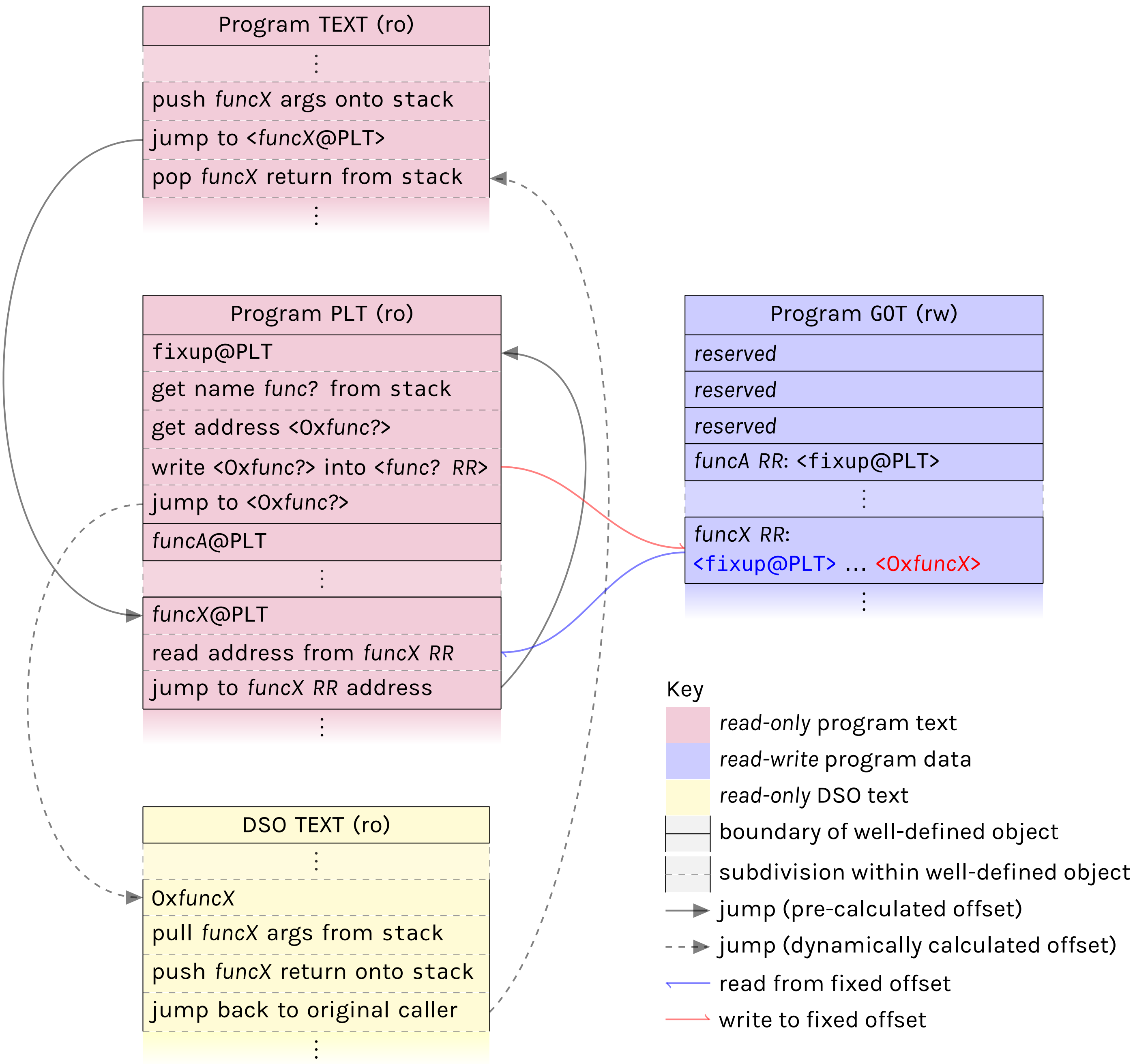
This is how the first call to a foreign function (whose address we do not yet know) looks:

- Calling code puts foreign function arguments on the stack
- Excution jumps to a fixed offset in the PLT (specific to this function)
- The PLT stub looks up the corresponding address in the GOT and jumps to it
- — The fixup code pointed at by the GOT asks the linker for the real address
- — The linker searches the calling DSOs dependencies for the symbol
- — The fixup code writes the address into the GOT slot
- — The fixup code jumps to the address in the GOT slot
- The code in the foreign DSO pulls the arguments off the stack
- The return value is pushed onto the stack
- Execution jumps back to the original DSO, just after the original call



Subsequent calls skip the stages marked — (in green) as the GOT slot will already contain the real symbol's address.

Drawing that out in greater detail:



Some things worth noting here (they'll be important later):

- Only the calling code in the DSO and the called code in the foreign DSO know anything about the arguments and return value (signature) of the called function. None of the intervening steps know or care about these things.
- The PLT is read-only, but the GOT is writable.
- The address stored in the GOT slot is the only thing we need to change to divert a function call.
- If we scribble an address into the GOT slot before the fixup code is ever called we can completely short circuit the linker's normal symbol search algorithms.

How the linker finds DSOs

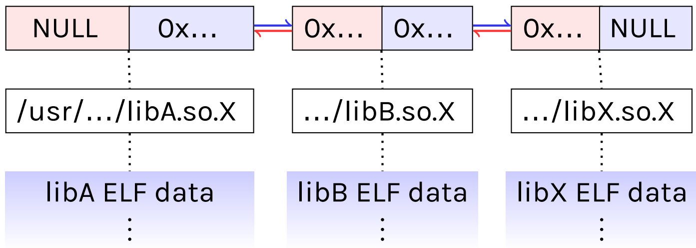
The observant reader may have noticed that we have glibly skimmed over some details above: "The linker searches the DSOs that the calling DSO depends on"

Let's unpack some of those details:

When the linker opens a DSO for the first time it scans it for items in the 'Dynamic Section'. There are many types of entry here, but for now we're interested in the `DT_NEEDED` entries. These entries tell the linker which DSOs are needed by the initial DSO.

The other piece of infrastructure we need to know about is the '**link map**' list. A link map list is a [doubly] linked list of currently loaded DSOs - when a DSO is linked its unresolved symbols are searched for in the link map list, but only DSOs mentioned explcitly in its `DT_NEEDED` entries, plus any DSOs which have been declared as globally available, are used for this search: Others are skipped over.

A simplified diagram showing a 3-element link map list:



Here's the rough process:

- Open the target DSO
- Scan it for `DT_NEEDED` entries
- For each dependency:
  - If the DSO is already mapped add its link map entry to the link map list for our target
  - Otherwise:
    - open the new DSO
    - map it into memory
    - add the new link map entry to the list for our target
    - recurse into this algorithm for the new DSO
- Add any declared-global DSOs to the target's link map list

NOTE: the `DT_NEEDED` entry is generally the bare DSO name - something like `libfoo.so.1` - and when checking the link map for a DSO it is this last part of the path that is compared. The exact path does not normally matter.

When the linker searches for a symbol for a given DSO, it searches only in the link map for the requesting DSO. (A given DSO can appear in multiple link maps, the same copy is re-used).

Now we understand how the linker searches for symbols - we've learned some important things:

- We have the foundations of restricted symbol visibility
- The key will be controlling which DSOs appear in our link map
- We can prevent a DSO from being loaded by preseeding a DSO with the same base name in the current link map

This looks pretty promising - we have all (or nearly all) the pieces of the puzzle. We might, in fact, have all the pieces if we're willing to get our hands very dirty indeed and hack up the link maps entirely by hand... But it turns out the glibc developers at Red Hat have already done some of the heavy lifting for us in the form of the `dlmopen()` call.

Introducing dlmopen()

First some background: The `dlopen()` C library call opens a DSO, in a similar way to the linker at program start up time: It opens the target and its `DT_NEEDED` dependencies (in the same way as the linker does, described above) and adds them to the link map for the new DSO.

This is close to what we want, but not quite there for two main reasons:

- `dlopen()` will re-use existing link map entries which match the target DSO's `DT_NEEDED`, which might pull in the 'wrong' version of a DSO.
- If a new DSO is later opened and flagged as global, and our DSO (or any of its `DT_NEEDED` entries) are also required by that new DSO, they will cease to be hidden and start being used by the linker (and we want to control DSO and symbol visibility).

This is where `dlmopen()` comes in: It is like `dlopen()`, but puts the new DSO's link map entries in a specific link map list (that you request) or creates a new list and puts the entries there.

If the new DSO mapping is not added to the default link map list the linker won't encounter it while resolving symbols for DSOs in the default namespace - this means no symbols from the new DSO will be exposed or used during normal linking.

Likewise since the new DSO gets its own private link map list, it won't (by default) have any of its symbols resolved from our main set of DSOs.

Now that we have all[1] the pieces of the puzzle, we can put together a solution to our problem.

[1] Spoiler: We're going to need more pieces.

Putting the pieces together

Let's start by stating the exact problem we're going to try and solve:

- We have a runtime in which we wish to start a program
- The runtime contains some, but not all of the DSOs we need
- The runtime appears to the program as a filesystem starting at /
- The real filesystem is bound into the runtime, and appears as `/host`
  - In other words, `/lib` from the OS appears as `/host/lib`, and so on.
- If a DSO comes from `/host`, we want to use only other DSOs from `/host`
- Only the explicitly required `/host` DSO should be used by the linker
  - ie anything else pulled in from `/host` should remain invisible to the program and other runtime DSOs
- The program should run completely unaltered: No rebuilds, no patching of the binary.

Supplying a *shim* DSO

The first step is allowing the program to find a DSO that satisfies its `DT_NEEDED` entry. We can't simply add the `/host` tree to the linker's search path (that wouldn't allow `/host` vs runtime segregation): We can, however, provide a fake library with the same name as our target DSO and let the linker find that.

This shim library is made easier to generate by the fact that we don't (as you may recall from earlier) need to know the signature of each function: We're never going to call the stubroutines it contains so we only need then to have a compatible entry in the symbol table of our shim DSO.

Loading and exposing the real symbols

Now that we've generated our fake DSO, we need to have it find the real symbols and patch up the GOTs of any DSOs that need them.

This requires a few steps:

- The list of symbols to export from the `dlmopen()` namespace is built into the library (it turns out this is the most reliable way to do this - and there's no reason not to other than code/binary size)
- The shim DSO harvests the `DT_NEEDED` entries recursively from the target DSO from `/host` and its dependencies, keeping track of which needs what
  - To find the relevant DSOs, our shim must read and interpret the linker cache at `/host/etc/ld.so.cache`, as well as searching `LD_LIBRARY_PATH`, if set, and also searching the default locations at `/lib` and `/usr/lib`
  - In addition to the path mapping we also have to be careful to ignore DSOs with the 'wrong' type - on x86* there are three common types: x86-64, i386 and x32. We only want to pick DSOs which are the same as the original DSO (the program)
  - While it does so it takes care to remap all paths to have a `/host` prepended to them, including resolving any symlinks manually with the extra `/host` element where necessary
  - Some DSOs should not be pulled from `/host`

- notably the linker (`ld.so`) itself. It also turns out that in order for the program to work reliably the libc cluster (the core `C library, libpthread and so on) must also be from the same tree as one another (and` ∴ not from /host)

- It then `dlmopen()`s the target DSOs in *reverse* dependency order: That is - Any DSOs with no dependencies are opened first, then the DSOs which only needed those first DSOs, and so on and so forth
  - The `dlmopen()` calls are made with the full `/host/…` path to the DSO in question, to make sure we pick up the /host version
  - The linker, when checking for `DT_NEEDED` items in the link map, only checks the base name of the DSO (the final `libfoo.X` part) so our `/host` DSOs only get other `/host` DSOs used to satisfy their requirements.
- Finally, once the full `/host` DSO set has been loaded, our shim DSO monkeypatches the runtime DSOs to make them use the right symbols (this is complicated enough that I'm going to break it out into its own section [see below]).

Monkeypatching a symbol into a DSO

ELF DSO layout in memory

This is where we need to dive into the details of how an ELF DSO is laid out in memory (as opposed to on disc) in even greater detail.

They key to all of this is the glibc library function `dl_iterate_phdr()`: it allows us to invoke a callback function of our choice on each DSO in the *default* namespace (it does *not* touch private `dlmopen()` namespaces).

Our callback function will know the base address in memory of the DSO, the name of the DSO, and the start of the array of ELF Program Headers.

There are many types of program header (see `/usr/include/elf.h`) but luckily for us we're only interested in `PT_DYNAMIC` sections. These are, unsurprisingly, the part(s) of the ELF DSO related to dynamic linking. There could technically be more than one dynamic section: We're going to iterate over them all and treat them all the same so this doesn't matter much.

In each `PT_DYNAMIC` section we find an array of `ElfW(Dyn)` entries, each with a `DT_…` type, finishing with an entry of type `DT_NULL` (see `elf.h` again).

These dynamic entries are not guaranteed to be in any particular order (although the `DT_*TAB` entries will appear at the start), and some of them can only be interpreted usefully by referring to the contents of other `DT_…` entries. Here are the ones we turn out to be interested in:

DT_STRTAB
The string table. This points to an area (within the current `PT_DYNAMIC` *section) that contains* `NUL` *terminated strings. Any* `DT_…` *item that needs to refer to a printable string will point to a location in this entry*

(pointer)

DT_SYMTAB
The symbol table. Contains some metadata about each symbol such as its visibility and type. Since we're working from a pre-compiled list we can assume most of these - we're mainly interested as a way of finding the name of the symbol in the `DT_STRTAB`

(pointer)

DT_PLTREL
This gives the type of the relocation table - `DT_REL` *or* `DT_RELA`

(integer)

DT_PLTRELSZ
The total size of the array of relocation records in the relocation table. Used for bounds-checking.

(integer)

DT_RELASZ/DT_RELSZ

The size of the DT_REL[A] *relocation table. At most one may be present (and must match the table type).*

(integer)

DT_JMPREL

The address of the relocation records. We need DT_PLTREL & DT_PLTRELSZ *to process this.*

(pointer)

DT_REL/DT_RELA

In addition to the relocation table pointed at by DT_JMPREL*, there may also be additional relocation tables existing independently. They're typically not interesting as all function-related relocations should be in the* DT_JMPREL *table, but we process them anyway as this layout rule is* 'just' a convention.

(pointer, not seen in testing so far)

Performing a manual symbol relocation

Having found the symtab, strtab and relocation array, we can process the entries there: Each relocation record contains the following information:

- The address (of the GOT entry, not the symbol) That is, the address of the address used by the PLT to call the function
- The relocation type. There are *many* relocation types (about 40 for x86-64 alone). Fortunately to relocate basic function calls we only need to handle a few per architecture: R_X86_64_JUMP_SLOT & R_386_JMP_SLOT being the most important - and occasionally R_X86_64_GLOB_DAT, R_X86_64_64 and their R_386... equivalents.

  [*Every relocation type has its own rules for how to calculate the correct jump.*]
- The addend (an additional offset to the address). This occurs if the item being relocated is an offset from another named object (eg if a variable is defined as a fixed offset into an array or struct, and the compiler knows this).

  [Only DT_RELA records have addends.]
- The index (0-based) of this symbol in the symtab.

Finally, having gathered all this information, we can perform a relocation (remember that we have to do this for every DSO from the runtime as each one will have its own GOT):

- Check the name of the symbol (relocation record → symtab → strtab) against the hard-coded relocation list in our shim DSO. If it's not there we're leaving this symbol alone.
- Get the real address of the symbol from the dlmopen() private namespace with dlsym().
- Calculate the address of the GOT slot which we wish to overwrite. To start with, it will contain the address of the fixup code in the PLT.
- Write the real address into the GOT slot.
- Proceed to beer island. We have monkeypatched one function symbol in one DSO!

What's left is repeating this process for every symbol in our hard-coded export list, for every DSO from the runtime.

Strictly speaking we should only monkeypatch symbols in DSOs which require the *real* library via a DT_NEEDED entry: If we are a shim for libfoo.so.1 then we should only try to replace symbols in other DSOs that explicitly depend on libfoo.so.1. In practice it has not been necessary to be careful about this (at least so far).

It's worth noting that we're using a very stupid, brute-force approach to matching each symbol: We don't use any of the clever hashing that the linker uses to speed things up. This is a deliberate ploy to keep the number of moving parts down to a minimum.

Lies, TODOs, and FIXMEs...

That's it, we're done. Except... for the lies. I have, for the sake of simplicity(!) told some lies and left out some details along the way. Here they are:

- Lie: The GOT slot contains the fixup address to start with.
- Lie: The GOT slot is (always) writable.

  These are sort of true. Mostly. Except that if the library is `RELRO` linked, the linker will pre-emptively fix up the addresses before any code is invoked by the DSO and then `mprotect()` the memory region containing the GOT to render it unwritable.

  The workaround is to read the `mprotect()`ed region data from `/proc/self/maps`, toggle the write bits on, do all our monkeypatching and then put them back the way they were.

- TODO: The symbol lookup has been simplified: I've left out all the details to do with symbol versioning.

  The initial goal of the project doesn't require symbol versioning support, so our shim DSOs currently don't handle it. At all.

- Lie: `dlopen()` will work in the private namespace

  Currently `dlopen()` cannot be called from any code that resides in a private `dlmopen()` namespace.

  There are two reasons for this: The project-related one is that our target library (`libGL`) uses `dlopen()` to open its drivers... but it doesn't know it's been re-mapped to `/host` and must get all its dependencies and modules from there.

  The workaround here is to monkeypatch all the DSOs *inside* the private namespace to divert `dlopen` to a wrapper that does all of the magic `dlmopen()` reverse-dependency-remap-to-host magic.

  Fortunately *most* of the pieces we need for this are ones we've already had to implement for monkeypatching DSOs from the runtime, with one glaring exception: `dl_iterate_phdr()` doesn't deliver private namespace entries to us.

  However we're in luck - we can use `dlinfo()` with `RTLD_DI_LINKMAP` to get a link map entry instead, and this turns out to have *almost* all the information that `dl_iterate_phdr` gives us (and all the info that we care about).

  [*The other problem is that* `dlopen()` *currently segfaults if called from inside a private* `dlmopen` *namespace - this is a glibc limitation*]

- LIE: Monkeypatching only needs to happen at start-up

  If the program ever `dlopen()`s a DSO after start up, and it has our target DSO in its `DT_NEEDED` entries, it will get the fake addresses from our shim DSO.

  The fix is to also wrap `dlopen()` in the runtime libraries, replacing it with a wrapper that calls the real `dlopen()` and then fixes up any new DSOs we haven't seen before.

- LIE: Monkeypatching is the only address-redirection we need to do

  If any of the runtime DSOs use `dlsym()` to find a symbol patched up by our shim library, they'll find the fake function instead.

  We wrap `dlsym()` in the same way as `dlopen()` aboveto handle this.

- FIXME: `dlmopen()` and `RTLD_GLOBAL` don't work together

  `dlopen()` supports `RTLD_GLOBAL`. `dlmopen()` does not. If the `/host` DSOs need `RTLD_GLOBAL` `dlopen()` calls to work then currently they... won't.

  This is another glibc limitation.

- TODO: We only need to worry about a few relocation types

  There are more relocation types we may need to support for this project to be generically useful. Variables (which don't go through the PLT) should be supported. Likewise TLS (Thread Local Storage) relocations will require more relocation type support.

  This is another item that isn't (I believe) particularly difficult, but it hasn't been necessary for the initial goals of our project, so onto the TODO list it goes.

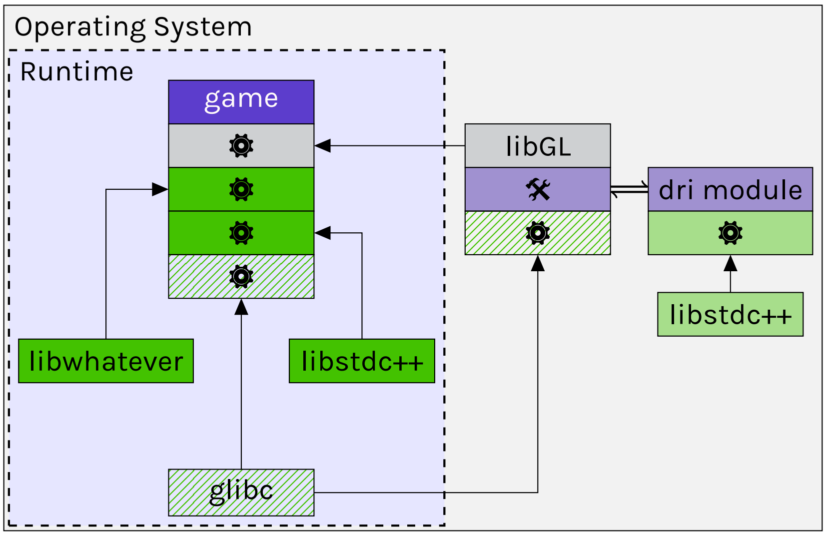
- LIE: The DSO namespaces can be completely segregated

  It turns out that for boring implementation detail reasons, some DSOs must not only be the same version but the same *instance* in both the main and secondary namespaces.

  In particular the `C library` cluster cannot function reliably with more than one copy, particularly if threading is activated at any time.

  This is not possible in any released `glibc` version, but patches implementing an `RTLD_SHARED` dlopen flag have been tried (successfully) and sent upstream for review.

This, then, is what our solution ends up looking like:



There you have it: Everything you never wanted to know about subverting the ELF dynamic linker.

Postscript: Putting it all into practice

To prove that all of this can actually be done the work-in-progress repo, which has been successfully used to proxy Mesa's `libGL`, can be found here:

libcapsule.git

www.collabora.com