libcapsule - Segregated Dynamic Linking

Vivek Das Mohapatra

Collabora

November 25, 2017
libcapsule - Segregated Dynamic Linking

Introduction

The Pieces of the Puzzle

These Yaks Aren't Shaving Themselves

And Finally

Open First
The Problem

- Applications ‘Containerised’
- Libraries come from a runtime

Runtime makes promises about API/ABI/versions, mesa tied to hw, no reasonable way for runtime to stay current.
The Problem

- Applications ‘Containerised’
- Libraries *mostly* come from a runtime
- ...but *some* still need to come from the host

runtime makes promises about API/ABI/versions mesa tied to hw, no reasonable way for runtime to stay current
The Problem

- Applications ‘Containerised’
- Libraries *mostly* come from a runtime
- …but *some* still need to come from the host
  - notably mesa (libGL & friends)
Applications ‘Containerised’

Libraries *mostly* come from a runtime

...but *some* still need to come from the host
  notably mesa (libGL & friends)

host libraries may have incompatible dependencies

runtime makes promises about API/ABI/versions mesa tied to hw, no reasonable way for runtime to stay current
What does the problem look like?

The linker uses sonames to decide if a library meets our requirements *but* sometimes we end up with incompatible libraries with the same soname… and it only allows one copy of the same soname in any link chain.
What could a solution look like?

We can see here two incompatible versions of libDEP from host and runtime: only libHOST sees the host version (and it does not see the runtime version).
Objectives

- Expose only the library we want to isolate
- its dependencies *not* exposed to us

---

libcapsule - Segregated Dynamic Linking

Introduction

Objectives

What do we mean by minimal intervention?
In order of preference:

- Purely runtime isolation mechanism
- Some compilation required but basically automatic
- Manual intervention required to generate the isolating ‘thing’
Objectives

- Expose only the library we want to isolate
- its dependencies *not* exposed to us
- No code changes

What do we mean by minimal intervention?
In order of preference:

- Purely runtime isolation mechanism
- Some compilation required but basically automatic
- Manual intervention required to generate the isolating ‘thing’
Objectives

- Expose only the library we want to isolate
- its dependencies not exposed to us
- No code changes
- No recompilation

What do we mean by minimal intervention?

In order of preference:

- Purely runtime isolation mechanism
- Some compilation required but basically automatic
- Manual intervention required to generate the isolating ‘thing’
Objectives

- Expose only the library we want to isolate
  - its dependencies *not* exposed to us
- No code changes
- No recompilation
- No performance hit

What do we mean by minimal intervention?

In order of preference:

- Purely runtime isolation mechanism
- Some compilation required but basically automatic
- Manual intervention required to generate the isolating ‘thing’
Objectives

- Expose only the library we want to isolate
  - its dependencies *not* exposed to us
- No code changes
- No recompilation
- No performance hit
- Transparent use of nonstandard search-path

What do we mean by minimal intervention?

In order of preference:

- Purely runtime isolation mechanism
- Some compilation required but basically automatic
- Manual intervention required to generate the isolating ‘thing’
Objectives

- Expose only the library we want to isolate
- its dependencies not exposed to us
- No code changes
- No recompilation
- No performance hit
- Transparent use of nonstandard search-path
- Minimal manual intervention

What do we mean by minimal intervention?

In order of preference:

- Purely runtime isolation mechanism
- Some compilation required but basically automatic
- Manual intervention required to generate the isolating ‘thing’
**libcapsule - Segregated Dynamic Linking**

**Introduction**

**The Pieces of the Puzzle**

These Yaks Aren’t Shaving Themselves

And Finally

Open First
1: Private Dependencies

- Library should have isolated dependencies
1: Private Dependencies

- Library should have isolated dependencies
- Normally all dependencies in a single linked list
1: Private Dependencies

- Library should have isolated dependencies
- Normally all dependencies in a single linked list
- So how do we do this?
- cf `dlopen()` but *can* create a new link map
- ... or can add a new entry to an existing link map
- more-or-less workable from at least glibc 2.19
2: Picking the right library versions

- dlmopen() automatically loads dependencies
2: Picking the right library versions

- dlmopen() automatically loads dependencies
- Libraries are picked from the search path
2: Picking the right library versions

- dlmopen() automatically loads dependencies
- Libraries are picked from the search path
- First match wins
2: Picking the right library versions

- `dlmopen()` automatically loads dependencies
- Libraries are picked from the search path
- First match wins
- We need to subvert this for isolated libraries
Controlling the link map

- Linker loads all listed dependencies

Effectively we populate the link map by hand - by doing dependency resolution by hand we prevent the linker’s automatic searching from kicking in: A classic convenience vs control trade-off.
Controlling the link map

- Linker loads all listed dependencies
- The linker won’t reload items already in the map

Effectively we populate the link map by hand - by doing dependency resolution by hand we prevent the linker’s automatic searching from kicking in: A classic convenience vs control trade-off
Controlling the link map

- Linker loads all listed dependencies
- The linker *won’t* reload items already in the map
  - Loading libraries explicitly
  - In reverse dependency order
  - We can control exactly what gets linked

Effectively we populate the link map by hand - by doing dependency resolution by hand we prevent the linker’s automatic searching from kicking in: A classic convenience vs control trade-off
3: Automatically exposing symbols

- Isolated the dlmopen()ed symbols completely
3: Automatically exposing symbols

- Isolated the dlmopen()ed symbols completely
- Need callers to get to them automatically
3: Automatically exposing symbols

- Isolated the dlmopen()ed symbols completely
- Need callers to get to them automatically
- Need to understand dynamic library calls
Jumping to a foreign function

Calling code puts foreign function arguments on the stack
Execution jumps to a fixed offset in the PLT (specific to this function)
The PLT stub looks up the corresponding address in the RR and jumps to it
First call - resolving the address

Library Text | Program Text | Stack | PLT | Linker | RR

- The fixup code pointed at by the RR asks the linker for the real address
- The linker searches the calling DSO dependencies for the symbol
- The fixup code writes the address into the RR slot
- The fixup code jumps to the address in the RR slot
The foreign function call

- Jump to funcX
- The code in the foreign DSO pulls the arguments off the stack
- Function does whatever it does
- The return value is pushed onto the stack
- Execution jumps back to the caller
If we scribble on the RR slot before the first call:
Control the RR, Control the call...

- If we scribble on the RR slot before the first call:
  - The PLT fixup will never be invoked

libcapsule - Segregated Dynamic Linking
- The Pieces of the Puzzle
  - Control the RR, Control the call...
Control the RR, Control the call...

- If we scribble on the RR slot before the first call:
  - The PLT fixup will never be invoked
  - The linker never resolves the symbol’s address
Control the RR, Control the call...

- If we scribble on the RR slot before the first call:
  - The PLT fixup will never be invoked
  - The linker never resolves the symbol’s address
  - Total control over where the function call goes

Key question — can we find the RR?

Yes — we can!

The link map → ELF data for each library
libelf can interrogate this
Control the RR, Control the call...

- If we scribble on the RR slot before the first call:
  - The PLT fixup will never be invoked
  - The linker never resolves the symbol’s address
  - Total control over where the function call goes
- Key question — can we find the RR?
Control the RR, Control the call...

- If we scribble on the RR slot before the first call:
  - The PLT fixup will never be invoked
  - The linker never resolves the symbol’s address
  - Total control over where the function call goes
- Key question — can we find the RR?
  - Yes — we can!
Control the RR, Control the call...

- If we scribble on the RR slot before the first call:
  - The PLT fixup will never be invoked
  - The linker never resolves the symbol’s address
  - Total control over where the function call goes
- Key question — can we find the RR?
  - Yes — we can!
  - The link map → to ELF data for each library
Control the RR, Control the call...

- If we scribble on the RR slot before the first call:
  - The PLT fixup will never be invoked
  - The linker never resolves the symbol’s address
  - Total control over where the function call goes

- Key question — can we find the RR?
  - Yes — we can!
  - The link map → to ELF data for each library
  - libelf can interrogate this
Putting the Pieces Together

- Make a shim library with the target’s soname
- Put the shim on the search path

Needs the list of exported symbols, but not their signatures
Needs to know its target’s soname
Otherwise fully automated
Putting the Pieces Together

- Make a shim library with the target’s soname
- Put the shim on the search path
- During the shim’s init:

Needs the list of exported symbols, but not their signatures
Needs to know its target’s soname
Otherwise fully automated
Putting the Pieces Together

- Make a shim library with the target’s soname
- Put the shim on the search path
- During the shim’s init:
  - `dlmopen()` the real library and its dependencies

Needs the list of exported symbols, but not their signatures
Needs to know its target’s soname
Otherwise fully automated
Putting the Pieces Together

- Make a shim library with the target’s soname
- Put the shim on the search path
- During the shim’s init:
  - `dlmopen()` the real library and its dependencies
    - Do this in reverse dependency order

Needs the list of exported symbols, but not their signatures
Needs to know its target’s soname
Otherwise fully automated
Make a shim library with the target’s **soname**

Put the shim on the search path

During the shim’s init:
  * `dlmopen()` the real library and its dependencies
  * Do this in reverse dependency order
  * Search the alternate library path
Putting the Pieces Together

- Make a shim library with the target’s **soname**
- Put the shim on the search path
- During the shim’s init:
  - `dlmopen()` the real library and its dependencies
  - Do this in reverse dependency order
  - Search the alternate library path
  - Walk the link map & scribble on all the RRs

Needs the list of exported symbols, but not their signatures
Needs to know its target’s soname
Otherwise fully automated
libcapsule - Segregated Dynamic Linking

Introduction

The Pieces of the Puzzle

These Yaks Aren’t Shaving Themselves

And Finally
Some Terminology

- Call an isolated set of libraries a *capsule*
- Assume they come from an fs mounted at */host*
dlopen() in capsules

- dlopen() can’t be called from inside a capsule
dlopen() in capsules

- dlopen() can’t be called from inside a capsule
- replace capsule’s dlopen with a wrapper that calls dlmopen()
- remap paths to /host in the wrapper
- dlmopen() doesn’t accept RTLD_GLOBAL
dlsym() now has a split personality

- dlsym() outside the capsule has to do extra work
dlsym() now has a split personality

- dlsym() outside the capsule has to do extra work
- Pretend that two separate dl handles are the same
dlsym() now has a split personality

- dlsym() outside the capsule has to do extra work
- Pretend that two separate dl handles are the same
- Do this when we scribble on the RRs
dlopen() outside capsules

- dlopen() outside capsule must trigger RR scribbling
dlopen() outside capsules

- dlopen() outside capsule must trigger RR scribbling
- replace external dlopen with a wrapper that does this
dlopen() outside capsules

- dlopen() outside capsule must trigger RR scribbling
- replace external dlopen with a wrapper that does this
- Again — when we scribble on the RRs
Extra Problems

- *alloc()/free() pairing

libcapsule - Segregated Dynamic Linking

- These Yaks Aren’t Shaving Themselves

Extra Problems
Extra Problems

- *alloc()/free() pairing
  - Propose RTLD\_SOMETHING

libcapsule - Segregated Dynamic Linking

These Yaks Aren’t Shaving Themselves

Extra Problems
∗alloc()/free() pairing
  ∗ Propose RTLDOMATIC
  ∗ For now, replace the ∗alloc/free cluster inside the capsule
libcapsule - Segregated Dynamic Linking

Introduction

The Pieces of the Puzzle

These Yaks Aren’t Shaving Themselves

And Finally

Open First
*drumroll*

Does it actually work?
”*drumroll*”

Does it *actually* work?

Yes!
*drumroll*

Does it *actually* work?

Yes!

- glxinfo et al
- openarena (SDL 1 & 2)
- Dungeon Defenders (SDL 2)
- And a Unity game whose name I forget...
libcapsule - Segregated Dynamic Linking

---

And Finally

Any Questions...?