


# Whirlwind tour of the Runtime Dynamic Linker

Goncalo Gomes

~ Software Development Engineer ~

~  ~ Big Data Infrastructure ~

# .init

## Agenda

1. Anatomy of a Linux C/C++ Library
2. Dynamic Link API
3. Linux Process Startup
4. Assorted DEMOS

The examples and this presentation will be available from

<https://github.com/gagomes/presentations>

# Who am I?

- Software Engineer at Workday, previously worked at Amazon and Citrix R&D
- Interested in various CS subjects, e.g.
  - Operating Systems, Virtualization, Debugging, Security, Algorithms, Compilers and Distributed Systems
- Wrote an ELF parsing library in 2002 and a small debugger to replace GDB
  - Eventually surrendered to using GDB

\*Not currently using C on a regular basis :sad\_panda:



# Libraries

- Libraries are reusable collections of data + code
- In Linux several other UNIX-like OSes, ELF is the default binary format for executables and libraries
- There are two main types of libraries in Linux
  - Static libraries
  - Dynamically libraries (aka DSO, or Dynamic Shared Objects)

# Static Libraries

- Statically linked libraries are a collection of object files bundled in an archive file
- **Pros**
  - Simple format. Think tarball of relocatable object files
  - Reduced link complexity (i.e each object file is bound to the executable file at link time)
  - Reduces runtime overhead
- **Cons**
  - Increases the final executable size
  - And consequently increases the memory footprint
  - Deployment of fixes for critical bugs and/or security vulnerabilities is harder

# Dynamic Libraries

- Dynamic linked libraries are a type of ELF file (**ET\_DYN**)
- **Pros**
  - Low storage and memory footprint
  - Can be loaded dynamically during runtime
  - Deployment of fixes for critical bugs and/or security vulnerabilities is easy
- **Cons**
  - Bindings occur during runtime
  - Symbol relocation is harder
  - Often requires PIC (Position Independent Code) in order to work

# Anatomy of a Linux C/C++ Library

## Sample “hello world” library

```
#include <stdio.h>

void hello(void) {
    printf("Hello, library world.\n");
}
```

## To build

```
gcc -fPIC -c -o helloworld.o helloworld.c
gcc -shared -o libhelloworld.so helloworld.o
```

# Library Build process

- Two distinct phases of compilation
- Step 1
  - Generate a relocatable object `helloworld.o`
- Step 2
  - Link objects in final assembly as a Shared Object (SO) aka `libhelloworld.so`



# Linking against libhelloworld.so

Invoking the `helloworld()` function from the `libhelloworld.so` library

```
#include "helloworld.h"

int main(int argc, char **argv)
{
    helloworld();
    return 0;
}
```

Building our sample application and linking it against `libhelloworld.so`

```
gcc -o app app.c -l helloworld -L .
```

# Dynamic Link API

- In libc since version 2.0 released in 1997
- Main header is `dlfcn.h`
- Consists of 4 main functions
  - `dlopen(3)`
  - `dlclose(3)`
  - `dlsym(3)`
  - `dlerror(3)`
- GNU extensions can be made visible by defining `_GNU_SOURCE` prior to inclusion of the header
- Requires explicit linking against **libdl**

# dlopen

```
#include <dlfcn.h>
```

```
void *dlopen(const char *filename, int flags);
```

- Opens the library pointed to by the filename parameter
- Typically called with one of **RTLD\_LAZY** or **RTLD\_NOW**
  - For additional flags, check the dlopen(3) man page
- On success returns a handle to the library
- On error NULL is returned and dlerror will be set

# dlclose

```
#include <dlfcn.h>
```

```
int dlclos(void *handle);
```

- Attempts to close a previously dlopen()'d handle
- On success returns 0, and on error returns a nonzero number

# dlsym

```
#include <dlfcn.h>
```

```
void *dlsym(void *handle, const char *symbol);
```

- Returns a pointer to the **symbol** from the specified handle
- The GNU extensions define a special handle, which points to the next loaded object containing the symbol name being requested
  - **RTLD\_NEXT**

# dlerror

```
#include <dlfcn.h>
```

```
char *dlerror(void);
```

- Returns a pointer to a string describing the most recent Dynamic Linking API usage related error

# Demo

# Caveats

- RTLD\_NEXT will find the next occurrence of a function in the search order after the current library. If the next library in the search order contains their own implementation of the symbol you're trying to use, it will default to that symbol.
- dl\*sym functions do not report errors via dlerror (fixed in glibc 2.24)
- GCC inlines some builtin functions
  - E.g: abs, cos, exp, fabs, fprintf, fputs, labs, log, memcmp, memcpy, memset, printf, putchar, puts, scanf, sin, snprintf, sprintf, sqrt, sscanf, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, vprintf and vsprintf
  - Compile with -fno-builtin and -O0



# Linux Process Startup - the 10,000 foot view

- This process entails:
  - System calls **fork** and **execve** are executed to spawn a new child process with our executable
  - The kernel then performs the following for us:
    - Identifies the type of executable file amongst a set of known / supported file formats
    - Sets up the environment variables and auxiliary vectors
    - Maps binary into memory
    - Locates the segment containing the interpreter (i.e [ld-linux.so](#))
    - Maps the interpreter into memory, freshens up the registers and jumps into the interpreter's entry **\_dl\_start**
      - **HERE BE DRAGONS**
    - Interpreter performs runtime dynamic linking operations (relocations and fixups)
    - it jumps into the executable's entry point, which typically means the **\_start**

# Special environment variables

- There are several linker related environment variables
  - LD\_LIBRARY\_PATH
  - LD\_DEBUG and LD\_DEBUG\_OUTPUT
  - LD\_PRELOAD
  - Etc.

# LD\_PRELOAD

- Loads and interposes a library in the chain of dependencies (link map)
- It's quite useful to subvert functionality from dynamic compiled executables without modifying them
- It's often used to superpose the functionality of existing code
  - e.g replace the calls to strcmp
- Some tricks it's widely used for:
  - When working a library where you may not want to recompile the executable every time
  - Replace the system malloc implementation with an alternative one
  - Make spotify play "Never gonna give you up" \*all\* the time
- For security reasons, it is ignored by setuid binaries, but can still be made available via [/etc/ld.so.preload](#)

# Demo

# Tools for managing and inspecting binaries

- Listing symbols - nm
- Listing dependencies - ldd
- Listing the ELF related information
  - eu-readelf from elf-utils (**recommended**) or alternatively readelf from binutils
- Managing library search paths and links - ldconfig

# References and sources

- Glibc Manual  
<https://www.gnu.org/software/libc/manual/>
- Linkers and Loaders  
<https://www.iecc.com/linker/>
- x86-64 ABI documentation (r252)  
<https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-r252.pdf>
- Ulrich Drepper's DSO howto  
<https://www.akkadia.org/drepper/dsohowto.pdf>
- Mayhem's RTLD internals document  
<http://s.eresi-project.org/inc/articles/elf-rtld.txt>
- Man pages: ld.so, dlopen, dlerror, dlsym, ar, gcc, ld, ldd, eu-readelf

Questions?

Thank you!