扫码订阅

泰晓科技，2021

目录

内容简介

Linux ELF 系列技术原创文章

笔者于 2019 年 11 月份左右，前后在泰晓科技连载 10 多篇 Linux ELF 相关的文章，内容涵盖 Linux 程序二进制格式、汇编语言、Linux 共享库工作原理、C 语言变量类型、ELF Sections 等。这些文章通过实例比较综合性地分析和展示了程序编译、链接、装载和运行的部分原理与实践。

这一系列文章不仅独立又很深度地介绍了 Linux 程序开发的某个知识点，也是《360° 剖析 Linux ELF》视频课程的很好补充。如果要系统而完整地学习和研究程序编译、链接、装载和运行的原理与实践，那么 订阅该课程 就显得非常有必要。

《360° 剖析 Linux ELF》视频课程

《360° 剖析 Linux ELF》 视频课程是一门提升计算机软件专业实践能力的必修课，也是一门毕业 3 年左右修炼内功的进阶课。

这门课程由 泰晓科技 推出，由具有 13 年以上 Linux 研发经验的吴章金老师主讲。吴老师是《C 语言编程透视》作者，官方 Linux 社区贡献者，Linux Lab、MIPS Ftrace 等开源项目作者，前 WindRiver 工程师，前魅族 Linux 部门技术总监。

该课程推出后获得很大反响，目前有数百名学员，听完课程的学员都给了积极反馈。该课程由四部分组成:

1. **8** 份幻灯片。内容充实、结构清晰、循序渐进、理论联系实际。
2. **10+** 小时视频。视频分辨率高、讲解耐心、可反复观看。
3. **70+** 份实验材料。实验材料丰富，并持续迭代更新，目标突破 100+ 份。
4. 课程专属微信答疑群。多位一线 Linux 技术专家驻群，及时精准地解答疑问，并鼓励开放性讨论。

课程详情:

类别	数量	备注	补充
幻灯片	8 份	全新设计了数十张图表	幻灯片通过电子邮件发送，请告知讲师邮箱地址
讲解视频	10+ 小时	高清画质，手机和PC可看	当前已经上线到 cctalk 平台
实验素材	70+ 多份	持续迭代和更新，5000多行	有专属代码仓库，需要注册 gitlab.com 帐号
演示小视频	10+ 多个	持续录制了十数个命令行视频	在代码仓库内，见所有 showterm.io 链接
技术文档	25+ 多份	整理了大量技术标准和规范	见代码仓库的子仓库 standards: make std
文章合集	10+ 篇	连载文章已制成 PDF 合集	见代码仓库 refs/360-linux-elf-v3.pdf
课程答疑群	1 个	多位技术专家驻群，响应及时	可联系微信号 tinylab
配套书籍	1 本	赠送《C 语言编程透视》PDF版	见代码仓库 refs/opencbook-v0.3.pdf

订阅课程，立即收获 **Debug** 核心技能

订阅地址：https://www.cctalk.com/m/group/88089283

关注公众号，勾搭一下吧

泰晓科技创建于 2010 年，是国内知名高质量 Linux 内容原创社区，聚拢了国内数百位 Linux 产品一线工程师和技术专家。

- 关注『泰晓科技』公众号，订阅更多 Linux 技术原创文章。
- 加微信号 tinylab，申请进 500 人 Linux 技术讨论群。

> 聚焦 Linux，追本溯源，见微知著！

微信扫码，即刻订阅《360°剖析Linux ELF》视频课程，与数百学员一起修炼 Linux

开发内力：

变更记录

2021.01.19

更新图文和购买链接，发布第 3 个合集。

2020.02.22

新增 4 篇 "ELF转二进制" 的文章系列，发布第 2 个合集。

2019.12.04

发布 『360° 剖析 Linux ELF』系列原创文章第 1 个合集。

微信扫码，即刻订阅《360°剖析Linux ELF》视频课程，与数百学员一起修炼 Linux 开发内力：

> By Falcon of TinyLab.org Aug 18, 2019

v5.1 开始剔除 a.out 格式

在 Linux 发布 5.1, Linux Lab 同步支持 一文中，首次得知了 Linux 移除 a.out 格式的消息，这个消息着实令人感叹，因为 a.out 伴随 Linux 的诞生至今在 Linux 中有将近 ~28 年的历史，而 a.out 本身则要追溯到更早的 Unix 时代。

下面是 v5.1 中两笔剔除 a.out 的动作：

```
$ git log --oneline v5.0..v5.1 | grep "a\.out"
eac6165 x86: Deprecate a.out support
08300f4 a.out: remove core dumping support
```

第 2 笔是 Linus 亲自改的，理由是 a.out 的 core dumping 功能年久失修了，而更进一步，因为 ELF 自 1994 年进入 Linux 1.0 以来，已经 ~25 年了，而且现在基本上找不到能产生 a.out 格式的编译器，所以 Borislav Petkov 直接在 x86 上把 `HAVE_AOUT` “干掉”，因此没法打开配置了。

a.out 核心代码还在

当然，大佬们做事还留有一点余地：

> Linux supports ELF binaries for ~25 years now. a.out coredumping has bitrotten quite significantly and would need some fixing to get it into shape again but considering how even the toolchains cannot create a.out executables in its default configuration, let's deprecate a.out support and remove it a couple of releases later, instead.

这意味着目前的 a.out 代码核心还在，想要用，把上面第 1 条变更 Revert 掉即可配置，可能涉及冲突要修复，如果嫌麻烦，对照 `git show eac6165` ，简单加回 `HAVE_AOUT` 即可：

```
$ git revert eac6165
```

笔者试着恢复以后，确实可以进行 a.out 的配置了。

准备一个支持 **a.out** 的内核

可以用 Linux Lab 来快速验证。

```
$ git clone https://gitee.com/tinylab/cloud-lab
$ cd cloud-lab
$ tools/docker/run linux-lab
```

上面的命令正常会拉起来一个浏览器，并自动登陆进一个 LXDE 桌面，进去后，打开控制台，即可参考 README.md 依次完成下述动作。

选择 `i386/pc` 作为测试板子：

```
$ make BOARD=i386/pc
```

并开始内核的下载、检出、配置、编译和运行：

```
$ make kernel-download
$ make kernel-checkout
$ make kernel-defconfig
$ make kernel-menuconfig
```

上述命令会启动配置，在配置里头打开：

> Executable file formats --->
>
> Kernel support for a.out and ECOFF binaries

接着完成编译并通过 nfsroot 启动：

```
$ make kernel
$ make boot ROOTDEV=/dev/nfs
```

通过 nfsroot 启动主要是方便后面在 qemu guest 和 qemu host 之间共享文件。这里也可以用 9pnet：

```
$ make boot SHARE=1
```

用 nfsroot 的话，共享目录 host 为 `boards/i386/pc/bsp/root/2019.02.2/rootfs`，guest 为根目录。

用 9pnet 的话，共享目录 host 为 `hostshare`，guest 为 `/hostshare`。也可以自行通过 `SHARE_DIR` 指定位置：

```
$ make boot SHARE=1 SHARE_DIR=$PWD/hostshare
```

尝试运行 **a.out** 格式

不过未来两三个版本以后，a.out 可能很快就被完全移除掉。

在 **a.out** 彻底被删除之前，来举行一个告别仪式吧：那就是尝试在 Linux v5.1 上跑一个真正的 a.out 格式的可执行文件。

既然 Linus 都说已经没有工具链默认能够生成 a.out 可执行文件，那怎么办呢?

你所看到的 **a.out** 并不是 **a.out** 格式

大家可能会说，gcc 默认编译生成的不就是 a.out 么？非也，此 a.out 非彼 a.out。

gcc 默认生成的 a.out 的实际格式是 ELF：

```
$ echo 'int main(void){ return 0; }' | gcc -x c - -
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dyn
amically linked, interpreter /lib64/l, for GNU/Linux 2.6.32, Bu
ildID[sha1]=baede5e2d5c16ba4b13a0e9d355acad2d237f6a7, not strip
ped
```

为什么 gcc 默认把 ELF 格式的可执行文件也默认取名为 a.out 呢？这个主要是历史沿革。

a.out 作为最早的可执行文件格式，其本意是 Assembler Output 的缩写。

虽然，现如今，汇编完还加入了链接环节，甚至还有动态链接环节，伴随着地是可执行文件格式从 a.out, COFF 到 ELF 一路演化下来，但是长久以来，这个 a.out 的默认名字却保留了下来。

gcc 不行，试试 objcopy 格式转换

既然现在的 gcc 默认不支持生成 a.out 格式，那尝试用 objcopy 转换看看，发现也不成功。

```
$ objcopy -O a.out-i386-linux a.out a.out-elf
objcopy: a.out-elf: can not represent section `.interp' in a.ou
t object file format
objcopy:a.out-elf[.interp]: Nonrepresentable section on output
```

既然不支持 `.interp`，那用一个不需要库函数的程序试试，Linux Lab 在 examples/assembly/x86 下提供了这样一个汇编程序。

```
$ cd (linux-lab)/examples/assembly/x86
$ make
$ objcopy -O a.out-i386-linux x86-hello x86-hello-a.out
$ file x86-hello-a.out
x86-hello-a.out: Linux/i386 demand-paged executable (ZMAGIC)
```

不幸地是，转换成功了，但是并不能执行。

```
$ ./x86-hello-a.out
./x86-hello-a.out: line 1: syntax error: unterminated quoted st
ring
```

这说明 ZMAGIC a.out 不知道哪天开始已经失效了。那 QMAGIC 呢，可是，目前没找到合适的方法强制转换为 QMAGIC 类型，欢迎读者们反馈补充。

关于 ZMAGIC 和 QMAGIC 的说明如下，摘自 A.OUT Manual page：

> OMAGIC The text and data segments immediately follow the header and are contiguous. The kernel loads both text and data segments into writable memory. NMAGIC As with OMAGIC, text and data segments immediately follow the header and are contiguous. However, the kernel loads the text into read-only memory and loads the data into writable memory at the next page boundary after the text. ZMAGIC The kernel loads individual pages on demand from the binary. The header, text segment and data segment are all padded by the link editor to a multiple of the page size. Pages that the kernel loads from the text segment are read-only, while pages from the data segment are writable.

试试 **Linux 0.11**

暂时还不该放弃努力，因为 a.out 是 Linux 一早就支持的格式，那为什么不试试 Linux 0.11，正好 Linux 0.11 Lab 还提供了一个可以在 Linux 0.11 上运行的编译器。

准备 **Linux 0.11 Lab**

Linux 0.11 Lab 可以直接在 Linux Lab 下跑，可以在 Linux Lab 中把它也 clone 到 `/labs` 目录下：

```
$ cd /labs
$ git clone https://gitee.com/tinylab/linux-0.11-lab
```

准备 **Hello.s**

接着先准备好一个可以在 Linux 0.11 编译的程序，可以直接用标准的 hello.c，也可以用汇编，这里直接复用上面的 `examples/assembly/x86/x86-hello.s`，把它复制到 Linux 0.11 的磁盘中，并稍作改动即可：

```
$ make mount-hd

$ sudo cp ../linux-lab/examples/assembly/x86/x86-hello.s rootfs
/_hda/usr/root/

$ sudo diff -Nubr ../linux-lab/examples/assembly/x86/x86-hello.
s rootfs/_hda/usr/root/x86-hello.s
--- ../linux-lab/examples/assembly/x86/x86-hello.s    2019-04-2
7 03:02:26.685203102 +0000
+++ rootfs/_hda/usr/root/x86-hello.s    2019-08-17 21:28:25.000
000000 +0000
@@ -1,12 +1,12 @@
 .data                   # section declaration
 msg:
-    .string "Hello, world!\n"
+    .ascii "Hello, world!\n"
     len = . - msg   # length of our dear string
 .text                   # section declaration
                         # we must export the entry point to th
e ELF linker or
-    .global _start      # loader. They conventionally recogniz
e _start as their
+    .globl _main        # loader. They conventionally recogniz
e _start as their
                         # entry point. Use ld -e foo to overri
de the default.
-_start:
+_main:
 # write our string to stdout
     movl    $len,%edx   # third argument: message length
     movl    $msg,%ecx   # second argument: pointer to message
to write

$ sync
$ make umount-hd
```

主要老版本 gcc 不支持 `.string`，需要用 `.ascii` 替换，另外默认入口需要改为 `_main`，而不再是 `_start`。

在 **Linux 0.11** 中编译 **Hello.s**

之后进入 Linux 0.11 Lab 并启动 Linux 0.11，这里选择从硬盘加载文件系统，其他文件系统不带编译器：

```
$ cd linux-0.11-lab
$ make boot-hd
```

启动以后，编译并验证一下：

```
$ gcc x86-hello.s
$ ./a.out
Hello, world!
```

如果需要编辑，记得通过 `mkdir /tmp` 创建一个 `/tmp` 目录，然后就可以用 vi 直接在 Linux 0.11 编辑代码了。

在 **Linux Lab** 中运行 **QMAGIC a.out**

之后，重新把磁盘挂起，通过 9pnet 指定相应目录为共享目录，这样就可以直接在 Linux 中运行了。

```
$ make mount-hd
$ ls rootfs/_hda/usr/root/
a.out x86-hello.s

$ sudo file rootfs/_hda/usr/root/a.out
rootfs/_hda/usr/root/a.out: a.out little-endian 32-bit demand p
aged pure executable not stripped

$ cd ../linux-lab
$ make boot SHARE=1 SHARE_DIR=$PWD/../linux-0.11-lab/rootfs/_hd
a/usr/root/
...
# /hostshare/a.out
fd_offset is not page aligned. Please convert program: a.out
Hello, world!
```

试试从 **a.out** 到 **ELF** 的转换

虽然上面的 ELF 转换成 ZMAGIC a.out，无法正常运行，但是反过来呢？试着从 QMAGIC a.out 转换为 ELF，竟然可以运行成功：

```
$ objcopy -O elf32-i386 a.out elf-hello
$ file ./elf-hello
elf-hello: ELF 32-bit LSB executable, Intel 80386, version 1 (S
YSV), statically linked, not stripped
$ sudo ./elf-hello
Hello, world!
```

补充一下，用 `objcopy --info` 可以列出支持的所有格式：

```
$ objcopy --info
       a.out-i386-linux pei-i386 pei-x86-64 elf64-l1om elf64-k1
om
  i386 a.out-i386-linux pei-i386 pei-x86-64 ---------- --------
--
  l1om ---------------- -------- ---------- elf64-l1om --------
--
  k1om ---------------- -------- ---------- ---------- elf64-k1
om
 iamcu ---------------- -------- ---------- ---------- --------
--
plugin ---------------- -------- ---------- ---------- --------
--
```

需要补充一点，上面如果直接运行 `./elf-hello`，会出现段错误，留待后续分解吧。

小结

到这里为止，经过诸多努力，终于在 a.out 彻底被从官方 Linux 剔除之前，完成了一次运行的尝试。

在这个基础上，未来，就有机会更深度地分析 a.out，COFF 到 ELF 三种格式以及它们的演进历程。

> By Falcon of TinyLab.org Aug 18, 2019

简介

ELF 在 Linux 系统中作为标准可执行文件格式已经存在了 ~25 年。

如果要在 Linux 下直接研究 ELF，通常很难绕过 Linux 本身的复杂度。

为了降低学习 ELF 的门槛，今天特地引荐一个极小的系统，这个小系统不仅有 “Bootloader”，有 “OS”，还能加载和运行标准的可以打印 Hello 的 ELF 程序。

所有这些都由 CS630-Qemu-Lab 提供。它是一套用于学习旧金山大学课程 CS 630: Advanced Microcomputer Programming 的极简实验环境，是一个由泰晓科技主导的开源项目。

它可以独立使用，也可以在 Linux Lab 下使用。下面介绍如何在 Linux Lab 下使用它。

准备环境

先准备 Linux Lab（非 Ubuntu 系统请提前安装好 Docker）：

```
$ git clone https://gitee.com/tinylab/cloud-lab
$ cd cloud-lab
$ tools/docker/run linux-lab
```

执行完正常会看到一个启动了 LXDE 桌面的浏览器，进去以后，点击桌面的控制台，启动后下载 CS630-Qemu-Lab 到 `/labs` 目录下：

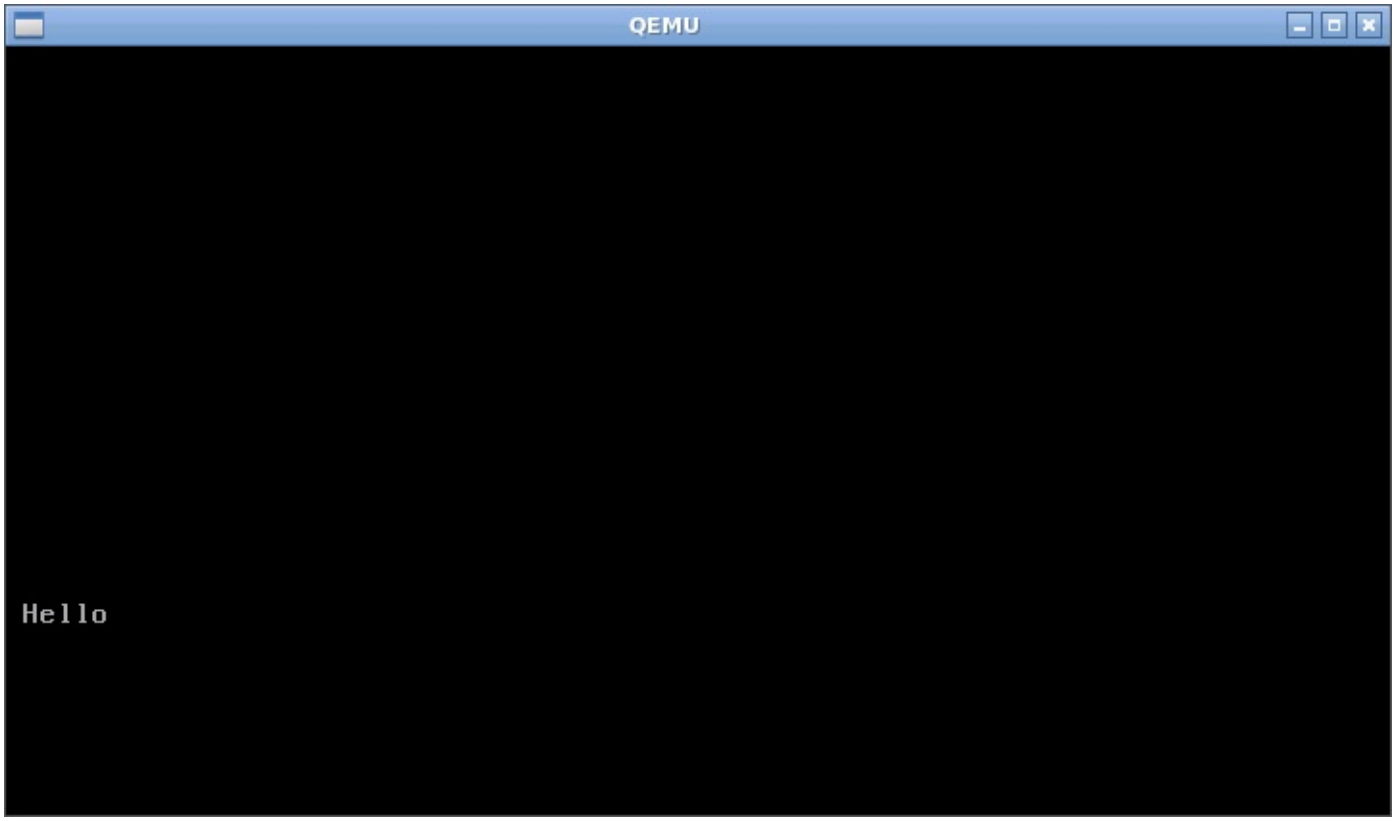
```
$ cd /labs
$ git clone https://gitee.com/tinylab/cs630-qemu-lab
$ cd cs630-qemu-lab
```

一键运行 **ELF**

接下来，通过 CS630 Qemu Lab 提供的极速体验方式，一键编译 Bootloader, OS 和 Hello 汇编程序，并自动依次运行：

```
$ make boot SRC=res/elfexec.s APP=res/hello.s
```

运行完以后，会弹出一个 Qemu 界面，并在屏幕打印一个 Hello 字符串。



默认是从软盘加载程序，如果要改为硬盘，可以用：

```
$ make boot-hd SRC=res/elfexec.s APP=res/hello.s
```

初步解读

这个实验主要包含如下三部分：

- “Bootloader”

    - `src/quikload_floppy.s`：实际代码只有 68 行
    - `src/quikload_hd.s`：实际代码只有 44 行
- “OS”
    - `res/elfexec.s`：实际代码只有 430 行，有提供 write, exit 等几个小的系统调用，还能加载标准 ELF
- “APP”
    - `res/hello.s`：实际代码仅 19 行，可以打印 Hello 字符串，编译生成标准的 ELF 程序

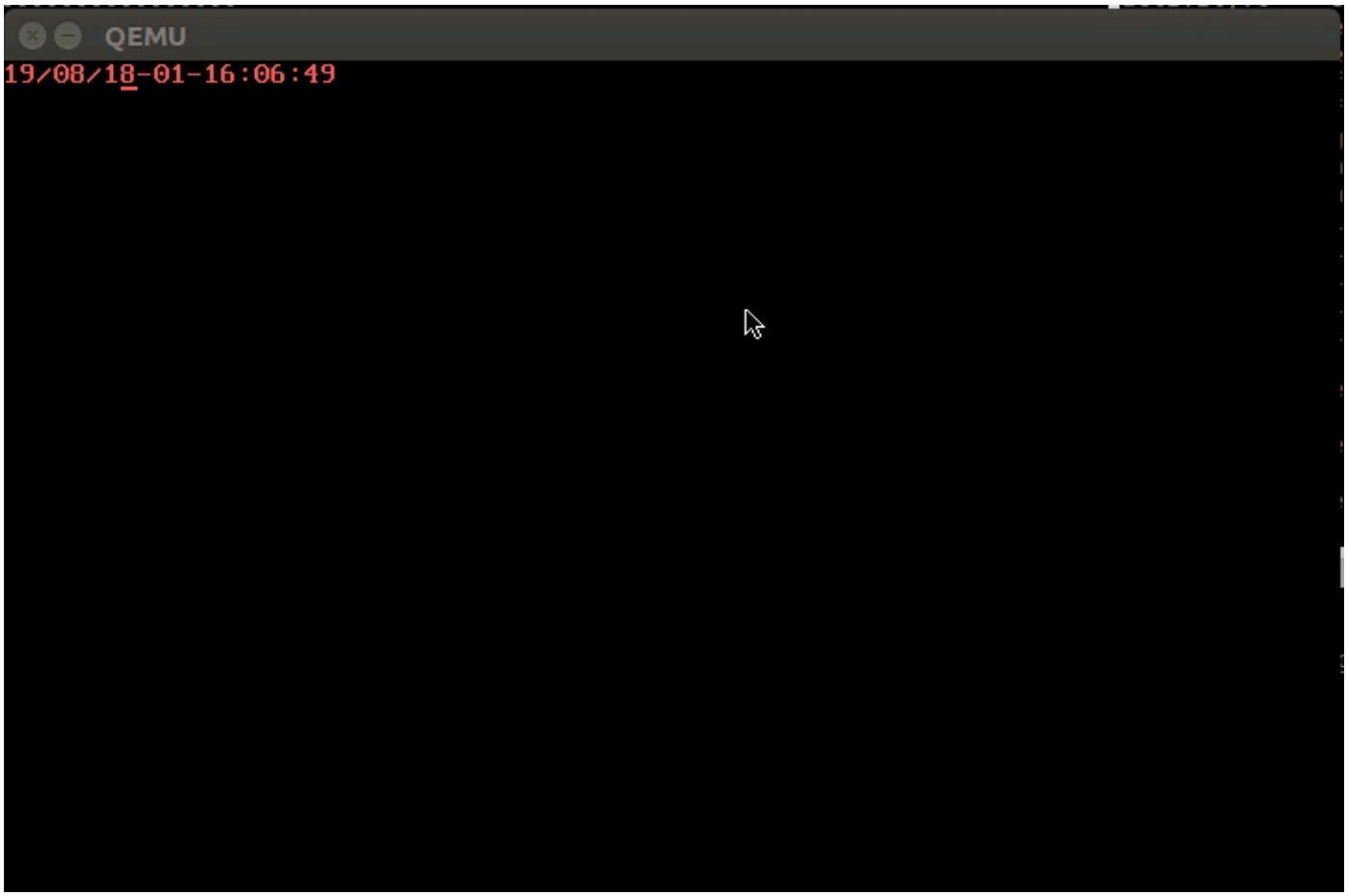
以 Floppy 版本为例，上述 bootloader 和 os 加起来仅有 498 行代码，含注释和空行也才 644 行，相比庞大的 Linux 来讲，可谓极其微小，因此特别适合核心 ELF 原理分析。

题外话

CS630 Qemu Lab 的汇编语言实验案例非常丰富，全部以 X86 为平台，以 Linux AT&T 汇编语法撰写，代码简洁清晰，非常适合学习。例如，跑一个 rtc 程序：

```
$ make boot SRC=src/rtc.s
```

下面是演示视频：



小结

Jonathan Blow 在莫斯科 DevGAMM 上，做了一个题为《阻止文明倒塌》的演讲。

这个演讲反应了一个普遍的情况，某个项目，随着功能的迭代和技术的发展，其功能不断丰富，复杂度却在不断增加。对于后来者，学习难度和门槛就变得越来越高。很多内容，由于逐步远离了当初设计者和开发者的环境，新来的维护人员极易出现理解偏差，随着老一辈 Maintainers 逐渐地离开，系统可能会变得越来越难以维护。

Linux 有点类似这样，它正在变得越来越复杂，我们学习一个可执行文件格式，得抱着几本大砖头 Linux 图书，并从数万行代码中找出那些关联的片段，看上去就是一个令人畏惧的工程。

泰晓科技致力于降低 Linux 技术的学习和研究门槛，我们正在做很多努力，去简化问题的复杂度，一方面构建了多套极简又容易快速上手的实验环境，另外一方面，从产品实战的细微处追本溯源，分享了大量的技术原创文章，争取见微知著。

后续将进一步深度解读这个 498 行的极小系统。

> By Falcon of TinyLab.org Aug 01, 2019

新版 gcc 默认开启了几个选项，导致学习汇编语言，尤其是入门的同学，很难。

以如下代码为例：

```
$ cat demo.c
#include <stdio.h>

int main(void)
{
    int i;
    char buffer[64];

    i = 1;
    buffer[0] = 'a';

    return 0;
}
```

下面这条指令可以生成比较干净简洁的代码：

```
$ gcc -fno-stack-protector -fomit-frame-pointer -fno-asynchrono
us-unwind-tables -S demo.c
```

结果如下：

```
$ cat demo.s
    .file    "demo.c"
    .text
    .globl    main
    .type    main, @function
main:
    movl    $1, -4(%rsp)
    movb    $97, -80(%rsp)
    movl    $0, %eax
    ret
    .size    main, .-main
    .ident    "GCC: (Ubuntu 8.3.0-16ubuntu3~16.04) 8.3.0"
    .section    .note.GNU-stack,"",@progbits
```

加个 `-m32` 参数就可以生成 32 位的:

```
$ gcc -fno-stack-protector -fomit-frame-pointer -fno-asynchrono
us-unwind-tables -m32 -S demo.c
$ cat demo.s
    .file    "demo.c"
    .text
    .globl    main
    .type    main, @function
main:
    subl    $80, %esp
    movl    $1, 76(%esp)
    movb    $97, 12(%esp)
    movl    $0, %eax
    addl    $80, %esp
    ret
    .size    main, .-main
    .ident    "GCC: (Ubuntu 8.3.0-16ubuntu3~16.04) 8.3.0"
    .section    .note.GNU-stack,"",@progbits
```

稍微做个解释:

- `-fno-stack-protector`：去掉 stack 保护，stack protector 用于检查 stack 是否被踩
- `-fomit-frame-pointer`：不用 fp 寄存器 rbp/ebp，直接用 stack 寄存器 rsp/esp 就好了
- `-fno-asynchronous-unwind-tables`：消除 .eh_frame section

`.eh_frame` 是 DWARF-based unwinding 用来实现 `backtrace()`，`__attribute__((__cleanup__(f)))`，`__buildtin_return_address(n)`，`pthread_cleanup_push` 等，具体请参考 assembly - Why GCC compiled C program needs .eh_frame...。现在无论是否用到这些功能，gcc 都加了 .eh_frame，所以不用的时候直接删除掉也无妨。

另外，Stack Protector 不是看上去的那么强大，从原理上看，如果刚好跳过了预设了值的位置去踩的话，Stack Protector 其实是检测不出来的，当然，有总比没有好。

下面这种是可以检测出来的：

```
$ cat demo.c
#include <stdio.h>

int main(void)
{
    char buffer[2];
    int i;

    i = 1;
    buffer[0] = 'a';

    buffer[3] = 'b';

    printf("hello.world");

    return 0;
}
```

编译和运行，确保可以生成 coredump：

```
$ gcc -o demo demo.c
$ ulimit -c unlimited
$ ./demo
*** stack smashing detected ***: ./demo terminated
hello.worldAborted (core dumped)
```

用 gdb 分析 coredump:

```
$ gdb demo core
[New LWP 89783]
Core was generated by `./demo'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00007f76507fc428 in __GI_raise (sig=sig@entry=6) at ../sy
sdeps/unix/sysv/linux/raise.c:54
54    ../sysdeps/unix/sysv/linux/raise.c: No such file or direc
tory.
(gdb) bt
#0  0x00007f76507fc428 in __GI_raise (sig=sig@entry=6) at ../sy
sdeps/unix/sysv/linux/raise.c:54
#1  0x00007f76507fe02a in __GI_abort () at abort.c:89
#2  0x00007f765083e7ea in __libc_message (do_abort=do_abort@ent
ry=1,
    fmt=fmt@entry=0x7f765095649f "*** %s ***: %s terminated\n")
 at ../sysdeps/posix/libc_fatal.c:175
#3  0x00007f76508e015c in __GI___fortify_fail (msg=&lt;optimize
d out&gt;, msg@entry=0x7f7650956481 "stack smashing detected")
    at fortify_fail.c:37
#4  0x00007f76508e0100 in __stack_chk_fail () at stack_chk_fail
.c:28
#5  0x00000000004005c0 in main ()
(gdb)
```

可以粗略定位到有 Stack Overflow 的函数，但不能定位到具体哪一行踩了数据。

本文的例子汇整在 Linux Lab: examples/c/hello。

By Falcon of TinyLab.org Sep 05, 2019

前言

文件的终极存储方式是一堆二进制（01）串，在这个基础上，如果内容都能按照 8 位的 ASCII 文本表达，那就是纯文本文件，用各种文本编辑工具处理即可，如果内容是结构化的程序数据，比如可执行文件，那么得从二进制层面去操作。

对于特定的结构化数据，一般都有配套的操作 API，比如说 ELF，有专属的 binutils, elfutils 等，本文主要介绍通用的二进制操作工具，并以 ELF 为例，比照介绍相应的专属工具，这些工具在 Ubuntu 中都可以直接安装。

准备工作

先准备一个具体的小汇编程序，这个作为本文二进制操作演示的材料。

```
# hello.s
#
# as --32 -o hello.o hello.s
# ld -melf_i386 -o hello hello.o
# objcopy -O binary hello hello.bin
#

    .text
.global _start
_start:
    xorl   %eax, %eax
    movb   $4, %al                  # eax = 4, sys_write(fd, ad
dr, len)
    xorl   %ebx, %ebx
    incl   %ebx                     # ebx = 1, standard output
    movl   $.LC0, %ecx              # ecx = $.LC0, the addr of
string
    xorl   %edx, %edx
    movb   $13, %dl                 # edx = 13, the length of .
string
    int    $0x80
    xorl   %eax, %eax
    movl   %eax, %ebx               # ebx = 0
    incl   %eax                     # eax = 1, sys_exit
    int    $0x80

    .section .rodata
.LC0:
    .string "Hello World\xa\x0"
```

把上面这份代码汇编、链接，并执行：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ ./hello
Hello World
```

仅保留代码和数据：

```
$ objcopy -O binary hello hello.bin
```

经过上面两步，得到了两个二进制文件，一个是 ELF 可执行文件 hello，另外一个是只包含了代码和数据的二进制文件 hello.bin。

二进制查看

比较常用的二进制读取工具有：

- hexdump
- xxd
- od

查看 **hello.bin**

以读取 `hello.bin` 为例，三者可以输出类似的数据样式：

```
$ hexdump -C hello.bin
00000000  31 c0 b0 04 31 db 43 b9  6d 80 04 08 31 d2 b2 0d  |1...1.C.m...1...|
00000010  cd 80 31 c0 89 c3 40 cd  80 48 65 6c 6c 6f 20 57  |..1...@..Hello W|
00000020  6f 72 6c 64 0a 00 00                              |orld...|
00000027

$ xxd -g 1 hello.bin
00000000: 31 c0 b0 04 31 db 43 b9 6d 80 04 08 31 d2 b2 0d  1...1.C.m...1...
00000010: cd 80 31 c0 89 c3 40 cd 80 48 65 6c 6c 6f 20 57  ..1...@..Hello W
00000020: 6f 72 6c 64 0a 00 00                             orld...

$ od -A x -t x1z hello.bin
000000 31 c0 b0 04 31 db 43 b9 6d 80 04 08 31 d2 b2 0d  >1...1.C.m...1...<
000010 cd 80 31 c0 89 c3 40 cd 80 48 65 6c 6c 6f 20 57  >..1...@..Hello W<
000020 6f 72 6c 64 0a 00 00                             >orld...<
000027
```

上面以字节流的方式一个一个打印出来，不用关心字节序，直接按照文件的字节存储顺序打印出来即可。

查看 **hello ELF** 的 **.text** 节区

下面通过上述工具从 `hello` ELF 中直接打印代码和数据。

首先，需要通过 binutils 提供的 `readelf -S` 先获取到代码段和数据段在文件中的偏移：

```
$ readelf -S hello
There are 6 section headers, starting at offset 0x130:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
 ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000
 00      0   0  0
  [ 1] .text             PROGBITS        08048054 000054 000019
 00  AX  0   0  1
  [ 2] .rodata           PROGBITS        0804806d 00006d 00000e
 00   A  0   0  1
  [ 3] .shstrtab         STRTAB          00000000 000105 000029
 00      0   0  1
  [ 4] .symtab           SYMTAB          00000000 00007c 000070
 10      5   3  4
  [ 5] .strtab           STRTAB          00000000 0000ec 000019
 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x
(unknown)
  O (extra OS processing required) o (OS specific), p (processo
r specific)
```

上面拿到了 `.text` 和 `.rodata` 两个 Section 在文件中的偏移（Off）和大小（Size）。

```
.text 0x54 0x19
.rodata 0x6d 0x0e
```

以 `.text` 为例，告知起始位置和长度就行了：

```
$ hexdump -C -s $((0x54)) -n $((0x19)) hello
00000054  31 c0 b0 04 31 db 43 b9  6d 80 04 08 31 d2 b2 0d  |1.
..1.C.m...1...|
00000064  cd 80 31 c0 89 c3 40 cd  80                       |..
1...@..|
0000006d

$ xxd -g 1 -seek $((0x54)) -l $((0x19)) hello
00000054: 31 c0 b0 04 31 db 43 b9 6d 80 04 08 31 d2 b2 0d  1...
1.C.m...1...
00000064: cd 80 31 c0 89 c3 40 cd 80                       ..1.
..@..

$ od -A x -t x1z -j $((0x54)) -N $((0x19)) hello
000054 31 c0 b0 04 31 db 43 b9 6d 80 04 08 31 d2 b2 0d  >1...1.
C.m...1...<
000064 cd 80 31 c0 89 c3 40 cd 80                       >..1...
@..<
00006d
```

使用 **ELF** 专属工具 **objdump**

针对 ELF，`objdump` 可以实现同样功能，并且更有针对性，其中 `-d` 为反汇编，`-j` 指定目标 Section：

```
$ objdump -d -j .text hello
hello:     file format elf32-i386


Disassembly of section .text:

08048054 <_start>:
 8048054:    31 c0                   xor    %eax,%eax
 8048056:    b0 04                   mov    $0x4,%al
 8048058:    31 db                   xor    %ebx,%ebx
 804805a:    43                      inc    %ebx
 804805b:    b9 6d 80 04 08          mov    $0x804806d,%ecx
 8048060:    31 d2                   xor    %edx,%edx
 8048062:    b2 0d                   mov    $0xd,%dl
 8048064:    cd 80                   int    $0x80
 8048066:    31 c0                   xor    %eax,%eax
 8048068:    89 c3                   mov    %eax,%ebx
 804806a:    40                      inc    %eax
 804806b:    cd 80                   int    $0x80
```

而 `-s` 则能输出类似 `hexdump` 等三样工具的输出格式:

```
$ objdump -s -j .text hello

hello:     file format elf32-i386

Contents of section .text:
 8048054 31c0b004 31db43b9 6d800408 31d2b20d  1...1.C.m...1...
 8048064 cd8031c0 89c340cd 80                 ..1...@..
```

hexdump 输出样式客制化

最后补充 `hexdump` 的更复杂功能，这个功能允许灵活调整数据显示样式，下面两个样式分别对齐 `xxd -g 1` 和 `od -A x -t x1z` 。

```
$ hexdump -v -e '"%08.8_ax: " 16/1 "%02x " "  "' -e '16/1 "%_p"
"\n"' hello.bin
00000000: 31 c0 b0 04 31 db 43 b9 6d 80 04 08 31 d2 b2 0d  1...
1.C.m...1...
00000010: cd 80 31 c0 89 c3 40 cd 80 48 65 6c 6c 6f 20 57  ..1.
..@..Hello W
00000020: 6f 72 6c 64 0a 00 00                            orld
...

$ hexdump -v -e '"%6.6_ax " 16/1 "%02x ""  "' -e '">"16/1 "%_p"
"<""\n"'  hello.bin
000000 31 c0 b0 04 31 db 43 b9 6d 80 04 08 31 d2 b2 0d  >1...1.
C.m...1...<
000010 cd 80 31 c0 89 c3 40 cd 80 48 65 6c 6c 6f 20 57  >..1...
@..Hello W<
000020 6f 72 6c 64 0a 00 00                            >orld..
.<
```

这意味着 `hexdump` 在数据展示上相比 `xxd` 和 `od` 更为灵活强大，很适合需要丰富样式的数据分析场景。

二进制编辑

大家推荐的 3 种二进制编辑工具：

- hexedit
- vim + xxd/xxd -r
- echo + dd

下面以一个具体例子来演示三者的用法，那就是直接在二进制中改掉要打印的字符串，把 'Hello World' 改为 'nihao world'。需要确保两者长度一样，不然就会把其他内容覆盖了。

使用 **hexedit**：支持 **Hex/ASCII** 两种模式

`hexedit` 支持十六进制和 ASCII 两种编辑模式，通过下面几步完成修改：

```
$ hexedit hello

1. TAB：切换 Hex 到 Ascii 模式
2. CTRL+S：搜索 Hello
3. 然后直接输入 `nihao world`, 覆盖掉 Hello World
4. CTRL+X：保存并退出
5. CTRL+C：退出不保存

$ ./hello
nihao world
```

`hexedit` 的用法不是很复杂，可以看看 `man hexedit` 。不过对于 vim 用户来说，有一个适应过程。

使用 **vim + xxd**：完美兼容 **vim**

接下来，用 `vim` 和 `xxd` 配合做编辑，这个是在 vim 中，调用 `xxd` 把文件转换为十六进制，然后编辑，之后再转换为二进制。遗憾地是，这种方式不支持直接编辑文本，需要编辑十六进制，当然，好处是可以直接在 vim 中使用。

这里演示把 'nihao world' 改回 'Hello World'，先要搜索到 'nihao world' 的十六进制：

```
$ echo "nihao world" | hexdump -C
00000000  6e 69 68 61 6f 20 61 62  63 64 65 0a              |ni
hao world.|
0000000c
$ echo "Hello World" | hexdump -C
00000000  48 65 6c 6c 6f 20 57 6f  72 6c 64 0a              |He
llo World.|
0000000c
```

然后，开启编辑过程，先切换为十六进制，找到 '6e 69 68 ...' 所在位置，替换为 '48 65

6c ...':

```
$ vim hello
:%!xxd -g 1
```

编辑完，用 `xxd -r` 转换回二进制，之后，保存退出即可完成编辑。

```
:%!xxd -r
```

保存完执行：

```
$ ./hello
Hello World
```

使用 **echo + dd**：方便自动化，无需交互

下面，使用更为直观的非交互式方式改写这个字符串，也就是用 `echo + dd` 来完成：

```
$ readelf -S hello | egrep ".rodata|Name"
[Nr] Name              Type            Addr     Off    Size   E
S Flg Lk Inf Al
[ 2] .rodata           PROGBITS        0804806d 00006d 00000e 0
0   A  0   0  1
$ echo 'nihao world' | dd of=hello bs=1 seek=$((0x6d)) count=$(
(0x0e)) conv=notrunc status=none
$ ./hello
nihao world
```

对于非纯字符，需要先获取十六进制编码，也可以对照 `man ascii` 查表：

```
$ echo -n "nihao world" | hexdump -v -e '"\\""x"1/1 "%02x"' ; e
cho
\x6e\x69\x68\x61\x6f\x20\x61\x62\x63\x64\x65
$ echo "\x6e\x69\x68\x61\x6f\x20\x61\x62\x63\x64\x65" | dd of=h
ello bs=1 seek=$((0x6d)) count=$((0x0e)) conv=notrunc status=no
ne
$ ./hello
nihao world
```

使用 **ELF** 专属工具 **objcopy**

最后，针对 ELF，也可以用专属工具 `objcopy` 来完成 `.rodata` section 的直接更新:

```
$ echo 'nihao world' > nihao.txt
$ objcopy --update-section .rodata=nihao.txt hello
$ ./hello
nihao world
```

二进制补丁

这里介绍 3 种二进制补丁制作和应用工具，分别是:

- rdiff
- bsdiff / bspatch
- git diff / apply

首先推出 `git diff/apply --binary`，不过这个仅限 git 仓库中使用，这里不做深入介绍。下面简单演示另外两组工具。

准备两个二进制文件

以上面用到的 `hello` 和 `hello.nihao` 为例，制作 patch 文件并打上 patch。

准备两个不同的二进制文件:

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ cp hello hello.nihao

$ echo 'nihao world' > nihao.txt
$ objcopy --update-section .rodata=nihao.txt hello.nihao

$ ./hello
Hello World
$ ./hello.nihao
nihao world
```

rdiff

制作差分 patch:

```
$ rdiff signature hello hello.sig
$ rdiff delta hello.sig hello.nihao hello.patch
```

打上差分 patch:

```
$ rdiff patch hello hello.patch hello.new
```

验证:

```
$ chmod a+x hello.new
$ ./hello.new
nihao world
```

bsdiff / bspatch

制作差分 patch：

```
$ bsdiff hello hello.nihao hello.patch
```

打上差分 patch：

```
$ bspatch hello hello.new hello.patch
```

验证：

```
$ chmod a+x hello.new
$ ./hello.new
nihao world
```

小结

至此，本文介绍了二进制文件的查看、编辑和补丁三大系列，共 9 套工具，也有介绍 ELF 专属工具 `readelf`，`objdump` 和 `objcopy`。

通用的二进制操作工具可以直接访问到字节层面，但是需要提前了解内容的结构规范。专属工具的话有封装好的 API 去访问一组字节，不过需要额外的编程语言和函数库支持。

前者适合临时高效的现场分析，后者适合产品层面的开发。

> By Falcon of TinyLab.org Nov 02, 2019

前言

前段时间，有多位同学在“泰晓原创团队”微信群聊到 C 语言相关的两个问题：

- 如何让共享库文件也可以直接执行
- 如何在可执行文件中用 `dlopen` 解析自身的函数

这两个需求汇总起来，可以大体理解为如何让一个程序既可以作为共享库，又能够直接运行。

这类需求在 Linux 下面其实很常见，比如 ld-linux.so 和 libc.so：

```
$ file /lib/i386-linux-gnu/ld-linux.so.2
/lib/i386-linux-gnu/ld-linux.so.2: symbolic link to ld-2.23.so
$ file /lib/i386-linux-gnu/ld-2.23.so
/lib/i386-linux-gnu/ld-2.23.so: ELF 32-bit LSB shared object, I
ntel 80386, version 1 (SYSV), dynamically linked
$ /lib/i386-linux-gnu/ld-2.23.so
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]

$ file /lib/i386-linux-gnu/libc.so.6
/lib/i386-linux-gnu/libc.so.6: symbolic link to libc-2.23.so
$ file /lib/i386-linux-gnu/libc-2.23.so
/lib/i386-linux-gnu/libc-2.23.so: ELF 32-bit LSB shared object,
 Intel 80386, version 1 (GNU/Linux), dynamically linked

$ /lib/i386-linux-gnu/libc.so.6
GNU C Library (Ubuntu GLIBC 2.23-0ubuntu11) stable release vers
ion 2.23, by Roland McGrath et al.
```

那如何做到的呢?

先来看看两类文件的区别

当前 Linux 下面的二进制程序标准格式是 ELF，这类格式可以用来表示 4 种不同类型的文件：

- 可重定位目标文件（.o），用于静态链接
- 可执行文件格式，用于运行时创建进程映像
- 共享目标文件（.so，共享库），协同可执行文件创建进程映像
- Core dump（core），运行过程中崩溃时自动生成，用于调试

我们来看中间两类：

- 可执行文件
  - 如果不引用外部库函数，那么所有符号地址是确定的，执行加载后可直接运行
- 共享库
  - 如果可执行文件用到外部库函数，那么需要通过动态链接器加载引用到的共享库并在运行时解析用到的相应符号

所以，前者和后者通常情况下不是独立存在的，是联合行动的，两者差异明显：

- 可执行文件有标准的 C 语言程序执行入口 `main`，而共享库则并没有这类强制要求
- 后者为了确保可以灵活被多个可执行文件共享，所以，符号地址在链接时是相对的，在装载时动态分配和计算符号地址

接下来做个实验具体看看两者的区别，准备一个“烂大街”的 hello.c 先：

```
#include <stdio.h>

int main(void)
{
    printf("hello\n");

    return 0;
}
```

先来编译为可执行文件（ `-m32` 用来生成采用 i386 指令集的代码）：

```
$ gcc -m32 -o hello hello.c
$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV)
, dynamically linked, interpreter /lib/ld-
$ ./hello
hello
```

再来编译为共享目标文件，并尝试直接执行它：

```
$ gcc -m32 -shared -fpic -o libhello.so hello.c
$ file libhello.so
libhello.so: ELF 32-bit LSB shared object, Intel 80386, version
 1 (SYSV), dynamically linked
$ ./libhello.so
Segmentation fault (core dumped)
```

直接执行失败，再试试如何生成一个可执行文件来加载运行它，这个是引用共享库的通常做法：

```
$ gcc -m32 -o hello.noc -L./ -lhello
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./ ./hello.noc
hello
```

通过实验，可以确认“正常”创建出来的共享库并不能够直接运行，而是需要链接到其他可执行文件中。

上述编译选项简介：

> -shared Create a shared library.
>
> -fpic Generate position-independent code (PIC) suitable for use in a shared library

让可执行文件可共享

接下来，好好研究一番。

先来看一个 gcc 直接支持的方式：

```
$ gcc -m32 -pie -fpie -rdynamic -o libhello.so hello.c
$ file libhello.so
libhello.so: ELF 32-bit LSB shared object, Intel 80386, version
 1 (SYSV), dynamically linked

$ ./libhello.so
hello

$ gcc -m32 -o hello.noc -L./ -lhello
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./ ./hello.noc
hello
```

确实可以执行，而且可以作为共享库链接到其他可执行文件中。

上述编译选项简介：

> -pie Produce a position independent executable on targets that support it.
>
> -fpie These options are similar to -fpic and -fPIC, but generated position independent code can be only linked into executables.
>
> -rdynamic Pass the flag -export-dynamic to the ELF linker, on targets that support it. This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table.

`-rdynamic` 等价于 `-Wl,-E` / `-Wl,--export-dynamic`，确保所有“库”中的符号都 export 到动态符号表，包括当前未用到的那些符号。

举个例子，如果 `hello.c` 有一个独立的 `hello()` 函数，没有别的函数（这里是指 `main`）调用到，但是其他用到该库的可执行文件希望用到它，那么 `-rdynamic` 就是必须的。

```
$ cat hello.c
#include <stdio.h>

void hello(void)
{
    printf("hello...\n");
}

int main(void)
{
    printf("hello\n");

    return 0;
}
$ cat main.c
#include <stdio.h>

extern void hello(void);

int main(void)
{
    hello();
    return 0;
}

$ gcc -m32 -pie -fpie -rdynamic -o libhello.so hello.c
$ readelf --dyn-syms libhello.so  | grep hello
    19: 00000662    43 FUNC    GLOBAL DEFAULT   14 hello

$ gcc -m32 -o hello.main main.c -L./ -lhello
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./ ./hello.main
hello...
```

如果没有 `-rdynamic`，链接时就没法使用。

```
$ gcc -m32 -o hello.main main.c -L./ -lhello
main.c:(.text+0x7): undefined reference to `hello'
```

同理，`dlopen` 自解析时也需要 `-rdynamic`：

```
#include <stdio.h>
#include <stdlib.h>
#define _GNU_SOURCE
#include <dlfcn.h>

void hello(void)
{
    printf("hello...\n");
}

int main(void)
{
    void *handle;
    void (*func)(void);
    char *error;

    handle = dlopen(NULL, RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        return EXIT_FAILURE;
    }

    dlerror();    /* Clear any existing error */

    func = (void (*)(void)) dlsym(handle, "hello");

    error = dlerror();
    if (error != NULL) {
        fprintf(stderr, "%s\n", error);
        return EXIT_FAILURE;
    }
```

```
    func();
    dlclose(handle);

    return 0;
}
```

实测效果：

```
$ gcc -m32 -pie -fpie -o libhello.so hello.c -ldl
$ ./libhello.so
./libhello.so: undefined symbol: hello

$ gcc -m32 -pie -fpie -rdynamic -o libhello.so hello.c -ldl
$ ./libhello.so
hello...
```

让共享库可执行

下面来探讨另外一种方式，在生成共享库的基础上，来研究怎么让它可以执行。

先来回顾一下共享库，在本文第 2 节直接执行的时候马上出段错误，基本原因是共享库没有强制提供一个标准的 C 程序入口。

即使是我们提供了 `main()` （把标准 hello.c 编译为 libhello.so），程序的入口并没有指向它。

```
$ readelf -h libhello.so | grep "Entry point"
  Entry point address:               0x3d0
$ objdump -d libhello.so | grep 3d0 | head -2
 380:    e8 4b 00 00 00           call   3d0 <__x86.get_pc_thun
k.bx>
000003d0 <__x86.get_pc_thunk.bx>:
```

那么，先解决入口的问题并运行，同样出错了：

```
$ gcc -m32 -shared -fpic -o libhello.so hello.c -Wl,-emain
$ readelf -h libhello.so | grep "Entry point"
  Entry point address:               0x4b9
$ objdump -d libhello.so | grep 4b9 | head -2
000004b9 <main>:
 4b9:    8d 4c 24 04              lea    0x4(%esp),%ecx

$ ./libhello.so
Segmentation fault
```

加上 `-g` 编译用 gdb 来看看原因：

```
$ gcc -m32 -g -shared -fpic -o libhello.so hello.c -Wl,-emain
$ objdump -d libhello.so | grep 4b9 | head -2
000004b9 <main>:
 4b9:    8d 4c 24 04             lea    0x4(%esp),%ecx

$ ulimit -c unlimited
$ ./libhello.so
Segmentation fault (core dumped)

$ gdb ./libhello.so core
Core was generated by `./libhello.so'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000003a6 in ?? ()
(gdb) bt
#0  0x000003a6 in ?? ()
#1  0xf77344e3 in main () at hello.c:5
(gdb) l hello.c:5
1    #include <stdio.h>
2
3    int main(void)
4    {
5        printf("hello\n");
6
7        return 0;
8    }
(gdb)
```

可以看到是执行 `printf` 的时候出错，说明库函数的解析出了问题，主动用动态连接器跑一下看看：

```
$ /lib/i386-linux-gnu/ld-2.23.so ./libhello.so
hello
Segmentation fault (core dumped)
```

哇哦，可以解析符号并打印了，不过最后还是崩溃了？

如果去分析 glibc 的 `__libc_start_main` 不难发现，我们还少调用一个标准退出函数，改造过后：

```
$ cat hello.c
#include <stdio.h>
#include <unistd.h>

void main(void)
{
    printf("hello\n");

    _exit(0);
}
```

再编译运行就没段错误了。再进一步，同样是分析 glibc，发现实际的入口函数并非 `main()`，而是 `_start`。

```
$ cat hello.c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    printf("hello\n");

    return 0;
}

void _start(void)
{
    int ret;

    ret = main();
    _exit(ret);
}
```

编译时连入口都不用指定了：

```
$ gcc -m32 -g -shared -fpic -o libhello.so hello.c
$ /lib/i386-linux-gnu/ld-2.23.so ./libhello.so
hello
```

也可以当共享库使用：

```
$ gcc -m32 -o hello.noc -L./ -lhello
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./ ./hello.noc
hello
```

稍微补充，完整的 `_start` 还需要处理参数、实现 init/fini 逻辑，这里限于篇幅不做展开。

最后还有一点遗憾，怎么样才能“动态”链接，而不是手动指定动态链接器呢？我们在程序中主动加入一个 `.interp` 节区来指定动态链接器吧。

```
$ cat hello.c
#include <stdio.h>
#include <unistd.h>

asm(".pushsection .interp,\"a\"\n"
    "        .string \"/lib/i386-linux-gnu/ld-linux.so.2\"\n"
    ".popsection");

int main(void)
{
    printf("hello\n");

    return 0;
}

void _start(void)
{
    int ret;

    ret = main();
    _exit(ret);
}
```

再试试，完美运行：

```
$ gcc -m32 -shared -fpic -o libhello.so hello.c
$ ./libhello.so
hello
```

最后，稍微整理一下：

```
$ cat hello.c
#include <stdio.h>

#ifdef EXEC_SHARED
```

```
#include <unistd.h>

asm(".pushsection .interp,\"a\"\n"
    "        .string \"/lib/i386-linux-gnu/ld-linux.so.2\"\n"
    ".popsection");

int entry(void)
{
    printf("%s %d: %s(): the real entry of shared library here.
\n", __FILE__, __LINE__, __func__);

    /* do whatever */

    return 0;
}

int main(void)
{
    return entry();

    return 0;
}

void _start(void)
{
    int ret;

    ret = main();
    _exit(ret);
}
#endif

void hello(void)
{
    printf("hello...\n");
}
```

当普通共享库使用，默认编译即可，要能够执行的话，实现一下 `entry()` ，编译时打开 `EXEC_SHARED` 即可：

```
$ gcc -m32 -shared -fpic -o libhello.so hello.c -DEXEC_SHARED
$ ./libhello.so
hello.c 12: entry(): the real entry of shared library here.

$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./ ./hello.main
hello...
```

小结

本文详细讲解了如何像 libc.so 和 ld-linux.so 一样，既可以当共享库使用，还能直接执行，并且讲述了两种方法。

两种方法都可以达成目标，第一种方法用起来简单方便，第二种方法揭示了很多背后的工作逻辑。

如果想与本文作者更深入地探讨共享库、程序执行、动态链接、内联汇编、程序真正入口等本文涉及的内容以及它们背后的程序链接、装载和运行原理，欢迎订阅吴老师的 10 小时 C 语言进阶视频课：《360° 剖析 Linux ELF》。

> By Falcon of TinyLab.org Nov 09, 2019

背景简介

如何创建一个可执行的共享库 一文谈完了如何让共享库可直接执行，本文再来谈谈共享库的运行时位置无关（PIC）是如何做到的。

> PIC = position independent code
>
> -fpic Generate position-independent code (PIC) suitable for use in a shared library

共享库有一个很重要的特征，就是可以被多个可执行文件共享，以达到节省磁盘和内存空间的目标：

- 共享意味着不仅磁盘上只有一份拷贝，加载到内存以后也只有一份拷贝，那么代码部分在运行时也不能被修改，否则就得有多个拷贝存在
- 同时意味着，需要能够灵活映射在不同的虚拟地址空间，以便适应不同程序，避免地址冲突

这两点要求共享库的代码和数据都是位置无关的，接下来先看看什么是“位置无关”。

什么是位置无关

同样以 hello.c 为例：

```
#include <stdio.h>

int main(void)
{
    printf("hello\n");

    return 0;
}
```

以普通的方式来编译并反汇编一个可执行文件看看：

```
$ gcc -m32 -o hello hello.c
$ objdump -d hello | grep -B1 "call.*puts@plt>"
 8048416:    68 b0 84 04 08           push   $0x80484b0
 804841b:    e8 c0 fe ff ff           call   80482e0 <puts@plt>
```

可以看到上面传递给 `puts`（printf）的字符串地址是“写死的”，在编译时就是确定的，这意味着 Load Address 也必须是固定的：

```
$ readelf -l hello | grep LOAD | head -1
  LOAD           0x000000 0x08048000 0x08048000 0x005b0 0x005b0
 R E 0x1000
```

上面可以看到 Load Address 为 0x8048000。

如果 Load Address 改变，数据地址就指向别的内容了，这就是“位置有关”。

共享库的话，必须摒弃这种“写死的”地址，要做到“位置无关”（注：prelink 是特殊需求，暂且不表）。

如何做到位置无关（**Part1**）

位置无关，意味着运行时可以灵活调整 Load Address，当 Load Address 在运行时发生改变后，代码还能被执行到，数据也能被正确访问。

那么代码和数据都变成跟 Load Address 相关的，不能再是绝对地址，而需要采用某个相对 Load Address 的地址。

动态链接器会负责找到可执行文件的共享库并装载它们，所以动态链接器是知道这个 Load Address 的，那么函数符号其实是很容易确定的，来看看不带 `-fpic` 时编译生成一个共享库：

- 查看 `main` 函数的初始地址

```
$ gcc -m32 -shared -o libhello.so hello.c
$ objdump -d libhello.so | grep -A 2 "main>:"
000004a9 <main>:
 4a9:    8d 4c 24 04               lea    0x4(%esp),%ecx
 4ad:    83 e4 f0                  and    $0xfffffff0,%esp
```

- 查看“装载地址”，编译后初始化为 0

```
$ readelf -l libhello.so | grep LOAD | head -1
  LOAD           0x000000 0x00000000 0x00000000 0x0057c 0x0057c
 R E 0x1000
```

- 确认 `main` 在文件中的偏移

```
$ readelf --dyn-syms libhello.so | grep m
Symbol table '.dynsym' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     4: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start
__
     9: 000004a9    46 FUNC    GLOBAL DEFAULT   11 main

$ hexdump -C -s $((0x4a9)) -n 10 libhello.so
000004a9  8d 4c 24 04 83 e4 f0 ff  71 fc                    |.L
$.....q.|
000004b3
```

可以看到，对于 `main` 而言，无论把共享库装载到哪里，动态链接器总能根据 Load Address 以及 `.dynsym` 中的偏移把 `main` 的运行时地址算出来（见 glibc：`_dl_fixup`）。

但是，这个时候（不用 `-fpic` 的话），数据地址和 puts 的地址也是“写死的”（实际上要到运行时才通过动态重定位确定运行时地址）：

```
$ objdump -d libhello.so  | grep -B1 "call.*main"
 4bd:    68 ec 04 00 00           push   $0x4ec
           4be: R_386_RELATIVE    *ABS*
 4c2:    e8 fc ff ff ff           call   4c3 <main+0x1a>
           4c3: R_386_PC32    puts@GLIBC_2.0
 4c7:    83 c4 10                 add    $0x10,%esp
```

作为对比，来看看加上 `-fpic` 的效果：

```
$ gcc -m32 -shared -fpic -o libhello.so hello.c
$ objdump -dr libhello.so  | grep -B6  "call.*puts@plt>"
 4c8:    e8 28 00 00 00           call   4f5 <__x86.get_pc_thun
k.ax>
 4cd:    05 33 1b 00 00           add    $0x1b33,%eax
 4d2:    83 ec 0c                 sub    $0xc,%esp
 4d5:    8d 90 10 e5 ff ff        lea    -0x1af0(%eax),%edx
 4db:    52                       push   %edx
 4dc:    89 c3                    mov    %eax,%ebx
 4de:    e8 bd fe ff ff           call   3a0 <puts@plt>
```

可以看到，用上 `-fpic` 以后，传递给 puts 的数据地址（ `push %edx` ）已经是通过动态计算的，那是怎么算的呢?

上面有个内联进来的函数很关键：

```
$ objdump -dr libhello.so  | grep -A3  "__x86.get_pc_thunk.ax>:
"
000004f5 <__x86.get_pc_thunk.ax>:
 4f5:    8b 04 24                 mov    (%esp),%eax
 4f8:    c3                       ret
```

这个函数贼简单，从栈顶取了一个数据就跳回去了，取的数据是什么呢? 这就要了解调用它的 `call` 指令了。

`call` 指令会把下一条指令的 `eip` 压栈然后 jump 到目标地址：

```
  call backward   ==>   push eip;
                        jmp backward
```

所以，数据地址是运行时计算的，跟运行时的 “eip” 给关联上了。

不难猜测，如果知道当前指令的位置，又提前保存了数据离当前位置的偏移，那么数据地址是可以直接计算的，只是上面那一段代码还是略微复杂了，因为有一堆 “Magic Number”。

不管怎么样，先来模拟计算一下，假设装载到的地址就是 0x0，那么执行到 `add` 指令时存到 eax 的 eip，恰好是 `call` 返回后下一条指令的地址，即 0x4cd：

```
 4c8:    e8 28 00 00 00           call   4f5 <__x86.get_pc_thun
k.ax>
 4cd:    05 33 1b 00 00           add    $0x1b33,%eax
 4d5:    8d 90 10 e5 ff ff        lea    -0x1af0(%eax),%edx
```

根据上述指令，那么 `%edx` 计算出来就是 0x510：

```
$ echo "obase=16;$((0x4cd+0x1b33-0x1af0))" | bc
510
```

再去取数据：

```
$ hexdump -C -s $((0x510)) -n 10 libhello.so
00000510  68 65 6c 6c 6f 00 00 00  01 1b                    |he
llo.....|
0000051a
```

果然是字符串的地址，所以，相对偏移其实被拆分成了两部分：`0x1b33` 和 `-0x1af0`。两个 "Magic Number" 一加就出来了。

所以，小结一下，“位置无关” 是通过运行时动态获取 “eip” 并加上一个编译时记录好的偏移计算出来的，这样的话，无论加载到什么位置，都能访问到数据。

如何做到位置无关（**Part2**）

这对 “Magic Number” 还是需要再看一看，既然是编译时确定的，看看汇编状态是怎么回事：

```
$ gcc -m32 -shared -fpic -S hello.c
$ cat hello.s | grep -v .cfi
...
.LC0:
    .string    "hello"
    .text
    .globl    main
    .type    main, @function
main:
.LFB0:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl    -4(%ecx)
    pushl    %ebp
    movl    %esp, %ebp
    pushl    %ebx
    pushl    %ecx
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    subl    $12, %esp
    leal    .LC0@GOTOFF(%eax), %edx
    pushl    %edx
    movl    %eax, %ebx
    call    puts@PLT
...
```

从 i386 的 archABI 不难找到这块的定义（P61~P62），`name@GOTOFF(%eax)` 直接表示 name 符号相对 %eax 保存的 GOT 的偏移地址。

首先，编译时要计算 `$_GLOBAL_OFFSET_TABLE` 和 `.LC0@GOTOFF` 。

`$_GLOBAL_OFFSET_TABLE_` 为 GOT 相对 `eip` 的偏移，可计算为：

> $*GLOBAL_OFFSET_TABLE* = .got.plt - eip

计算过程如下：

```
$ readelf -S libhello.so  | grep .got.plt
  [21] .got.plt          PROGBITS        00002000 001000 000010
 04  WA  0   0  4
$ echo "obase=16;$((0x2000-0x4cd))" | bc
1B33
```

接着，计算 `.LC0@GOTOFF` ：

> .LC0 - eip = $*GLOBAL_OFFSET_TABLE* + .LC0@GOTOFF .LC0@GOTOFF = .LC0 - eip - $*GLOBAL_OFFSET_TABLE*

计算过程如下：

```
$ echo "obase=16;$((0x510-0x4cd-0x1B33))" | bc
-1AF0
```

反过来，运行时的计算公式为：

> .LC0 = $*GLOBAL_OFFSET_TABLE* + .LC0@GOTOFF + eip .LC0 = 0x1B33 + (-1AF0) + eip
>
> .got.plt = $*GLOBAL_OFFSET_TABLE* + eip .got.plt = 0x1B33 + eip

实际上，只有 .got.plt 的地址，即 `ebx` 需要 `$_GLOBAL_OFFSET_TABLE_` 来计算，这个是用来做动态地址重定位的，暂且不表。

`.LC0` 的地址，完全可以换一种方式，直接用 `.LC0` 到 eip 的偏移即可，汇编代码改造完如下：

```
    call    __x86.get_pc_thunk.ax
.eip:
    # 计算 eip + (.LC0 - .eip) 刚好指向内存中的数据 "hello" 所在位置
    movl    %eax, %ebx
    leal    (.LC0 - .eip)(%eax), %edx

    # 计算 .got.plt 地址，_GLOBAL_OFFSET_TABLE_ 是相对 eip 的偏移，所以必须加上这个 offset：. - .eip
    addl    $_GLOBAL_OFFSET_TABLE_ + [. - .eip], %ebx
    subl    $12, %esp
    pushl   %edx
    call    puts@PLT
```

验证结果：

```
$ gcc -m32 -g -shared -fpic -o libhello.so hello.s
$ gcc -m32 -g -o hello.noc -L./ -lhello
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./ ./hello.noc
hello
```

小结

本文详细介绍了 Linux 下 C 语言共享库“位置无关”（PIC）的核心实现原理：即用 EIP 相对地址来取代绝对地址。

“位置无关” 代码会带来很大的内存使用灵活性，也会带来一定的安全性，因为“位置无关”以后就可以带来加载地址的随机性，给代码注入带来一定的难度。

由于有上述好处，各大平台的 gcc 都开始默认打开可执行文件的 `-pie -fpie` 了，因为 gcc 编译时开启了：`--enable-default-pie`。这也可能导致一些“衰退”，大家可以根据需要关闭它：`-no-pie`，`-fno-pie`。

当然，共享库的实现精髓不止于此，最核心的还是函数符号地址的动态解析过程，而这些则跟上面的 `.got.plt` 地址密切相关，受限于篇幅，暂时不做详细展开。

如果想进一步了解共享库动态链接的工作原理，进一步了解 `.got.plt` 并理解相关的过程链接表、全局偏移表的工作机制，欢迎订阅吴老师的 10 小时 C 语言进阶视频课：《360° 剖析 Linux ELF》。

> By Falcon of TinyLab.org Nov 15, 2019

背景介绍

Section 是 Linux ELF 程序格式的一种核心数据表达方式，用来存放一个一个的代码块、数据块（包括控制信息块），这样一种模块化的设计为程序开发提供了很大的灵活性。

需要增加一个功能，增加一份代码或者增加一份数据都可以通过新增一个 Section 来实现。Section 的操作在 Linux 内核中有着非常广泛的应用，比如内核压缩，比如把 .config 打包后加到内核映像中。

下面介绍三种新增 Section 的方式：汇编、C 和 ELF 工具。

通过内联汇编新增一个 **Section**

如何创建一个可执行的共享库 中有一个很好的例子：

```
asm(".pushsection .interp,\"a\"\n"
    "        .string \"/lib/i386-linux-gnu/ld-linux.so.2\"\n"
    ".popsection");
```

通过上述代码新增了一个 `.interp` Section，用于指定动态链接器。简单介绍一下这段内联汇编：

- `asm` 括号内就是汇编代码，这些代码几乎会被“原封不动”地放到汇编语言中间文件中（hello.s）。
- 这里采用 `.pushsection`，`.popsection`，而不是 `.section` 是为了避免之后的代码或者数据被错误地加到这里新增的 Section 中来。
- `.pushsection .interp, "a"`，这里的 "a" 表示 Alloc，会占用内存，这种才会被加到程序头表中，因为程序头表会用于创建进程映像。
- `.string` 这行用来指定动态链接器的完整路径。

稍微延伸两点：

- `.string` 可以替换为 `.incbin file`，然后把字符串内容放到名为 file 的文件中。文件末尾记得改为 `\0` 字节，可以用二进制编辑工具修改。
- `.string` 还可以替换为 `.ascii`，不过呢，末尾得主动加个 `\0` 字节，用法如下：

```
".ascii \"/lib/i386-linux-gnu/ld-linux.so.2\\x00\"\n"
```

`.incbin` 方式在 Linux 内核中用处相当广泛，例如：

- `arch/arm/boot/bootp/kernel.S: .incbin "arch/arm/boot/zImage"`
- `kernel/configs.c: .incbin \"kernel/config_data.gz\"`

本节完整演示代码如下：

```
#include <stdio.h>
#include <unistd.h>

#if 1
asm(".pushsection .interp,\"a\"\n"
    "        .ascii \"/lib/i386-linux-gnu/ld-linux.so.2\\x00\"\
n"
    ".popsection");

    /* .ascii above equals to .string \"/lib/i386-linux-gnu/ld-
linux.so.2\"\n */
#else
asm(".pushsection .interp,\"a\"\n"
    "        .incbin \"interp.section.txt\"\n"
    ".popsection");
#endif

int main(void)
{
        printf("hello\n");

        return 0;
}

void _start(void)
{
        int ret;

        ret = main();
        _exit(ret);
}
```

编译和执行：

```
$ gcc -m32 -shared -fpic -o libhello.so hello.c
$ ./libhello.so
hello
```

通过 gcc `__attribute__` 新增一个 Section

上面的需求可以等价地用 gcc `__attribute__` 编译属性来指定：

```
const char interp[] __attribute__((section(".interp"))) = "/lib
/i386-linux-gnu/ld-linux.so.2";
```

本节完整演示代码如下：

```
#include <stdio.h>
#include <unistd.h>

const char interp[] __attribute__((section(".interp"))) = "/lib
/i386-linux-gnu/ld-linux.so.2";

int main(void)
{
        printf("hello\n");

        return 0;
}

void _start(void)
{
        int ret;

        ret = main();
        _exit(ret);
}
```

编译和执行方法同上，不做重复介绍。

通过 **objcopy** 把某文件内容新增为一个 **Section**

上面介绍了 C 和汇编层面的方法，再来介绍一个工具层面的方法。

objcopy 这个工具很强大，其中就包括新增 Section。

接下来先准备一个文件 interp.section.text，记得末尾加的 `\0` 字节：

```
$ echo -e -n "/lib/i386-linux-gnu/ld-linux.so.2\x00" > interp.s
ection.txt
```

接着准备一个 hello.c，里头不指定任何 `.interp` ：

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
        printf("hello\n");

        return 0;
}

void _start(void)
{
        int ret;

        ret = main();
        _exit(ret);
}
```

不过需要注意的是，objcopy 不能直接在最终的可执行文件和共享库中加入一个 Section：

```
$ objcopy --add-section .interp=interp.section.txt --set-sectio
n-flags .interp=alloc,readonly libhello.so
objcopy:stTyWnxc: can't add section '.interp': File in wrong fo
rmat
```

怎么办呢，需要先加入到 `.o` 中，再链接，类似这样：

```
$ gcc -m32 -shared -fpic -c -o hello.o hello.c
$ objcopy --add-section .interp=interp.section.txt --set-sectio
n-flags .interp=alloc,readonly hello.o
$ gcc -m32 -shared -fpic -o libhello.so hello.o
```

注意，必须加上 `--set-section-flags` 配置为 alloc，否则，程序头会不纳入该 Section，结果将是缺少 INTERP 程序头而无法执行。

需要补充的是，本文介绍的 `.interp` 是一个比较特殊的 Section，链接时能自动处理，如果是新增了一个全新的 Section 类型，那么得修改链接脚本，明确告知链接器需要把 Section 放到程序头的哪个 Segment。

通过 **objcopy** 更新某个 **Section**

以上三种新增 Section 的方式适合不同的需求：汇编语言、C 语言、链接阶段，基本能满足日常的开发需要。

再补充一种方式，举个例子，上面用到的动态链接器来自 `libc6:i386` 这个包：

```
$ dpkg -S /lib/i386-linux-gnu/ld-linux.so.2
libc6:i386: /lib/i386-linux-gnu/ld-linux.so.2
```

如果系统安装的是 `libc6-i386` 呢?

```
$ dpkg -S /lib32/ld-linux.so.2
libc6-i386: /lib32/ld-linux.so.2
```

两个包提供的动态链接器路径完全不一样，那就得替换掉动态编译器，要重新编译 C 或者汇编吗?

其实不需要重新编译，因为可以直接这样换掉：

```
$ objcopy --dump-section .interp=interp.txt libhello.so
$ sed -i -e "s%/lib/i386-linux-gnu/ld-linux.so.2%/lib32/ld-linu
x.so.2%g" interp.txt
$ objcopy --update-section .interp=interp.txt libhello.so
$ ./libhello.so
hello
```

上面几组指令先把 `.interp` Section 取出来存到 interp.txt 文件中，再替换掉其中的动态链接器路径，最后再把新文件的内容更新进共享库。

小结

以上主要介绍了 Linux ELF 核心数据表达方式 Section 的多种 add 和 update 用法，掌握这些用户可以利于理解 Linux 内核源码中类似的代码，也可以用于实际开发和调试过程去解决类似的需求。

如果想与本文作者进一步探讨 Linux ELF 中各种类型 Section 的作用、更多操作方式（查看、裁剪、修改等）以及链接脚本的用法，欢迎订阅吴老师已经全面上线的 10 小时视频课程 《360° 剖析 Linux ELF》 。

> By Falcon of TinyLab.org Nov 24, 2019

背景说明

前几天，有同学在 “泰晓原创团队” 讨论群问道：

> 请教下，谭 C，8.9.3，用 static 声明静态局部变量，在实际中可有案例。

看到这个问题，立即浮现的概念是 RUN ONCE，内核源码找了一下：

```
$ grep -i "static.*run_once" -ur ./ --include "*.c"
./arch/mips/mm/page.c:  static atomic_t run_once = ATOMIC_INIT(
0);
./arch/mips/mm/page.c:  static atomic_t run_once = ATOMIC_INIT(
0);
./arch/mips/mm/tlbex.c: static int run_once = 0;
```

代码示例1:

```
void build_clear_page(void)
{
  static atomic_t run_once = ATOMIC_INIT(0);

  if (atomic_xchg(&run_once, 1)) {
    return;
  }
  /* body */
}
```

代码示例2:

```
void build_tlb_refill_handler(void)
{
  ...
  if (!run_once) {
    /* body */
    run_once++;
  }
}
```

另外，他又继续问道：

> 一般是用 static 定义一个全局的，很少看到函数内部在用 static?

这个问题则上升到 C 语言关键字 static 的用法。

static 这个关键字用来限定某个变量或者函数的作用域，这个作用域可能是文件层面，也可能是函数层面。

从语法的角度去解释某个关键字用法的文章很多，可是这些解释蛮多时候是很生硬的，不是那么好记忆。

本文尝试从实操的角度去解析 static 以及更多类型的 C 语言变量的形态。

从编译的角度

假如某个功能需求由多个文件构成如下：

```
$ cat print.h
extern void print(char *str);
extern char *hello;

$ cat hello.c
#include "print.h"

int main(void)
{
  print(hello);
  return 0;
}

$ cat print.c
#include <stdio.h>

char *hello = "hello";

void print(char *str);
{
  printf("%s\n", str);
}
```

编译运行如下：

```
$ gcc -m32 -o hello x.c print.h print.c
$ ./hello
hello
```

类似这种需要跨文件访问的函数和变量，如果定义成 static 的话：

```
$ cat print.c
#include <stdio.h>

static char *hello = "hello";

static void print(char *str);
{
  printf("%s\n", str);
}

$ gcc -m32 -o hello x.c print.h print.c
/tmp/ccetJaG2.o: In function `main':
x.c:(.text+0x12): undefined reference to `hello'
x.c:(.text+0x1b): undefined reference to `print'
collect2: error: ld returned 1 exit status
```

从 **ELF** 二进制程序文件的角度

先来编译成一个中间的可重定位文件：

```
$ gcc -m32 -c -o print.o print.c
```

针对加 `static` 的情况：

```
$ readelf -s print.o | egrep "hello$|print$"
     6: 00000000     4 OBJECT  LOCAL  DEFAULT    3 hello
     7: 00000000    23 FUNC    LOCAL  DEFAULT    1 print
```

不加的情况：

```
$ readelf -s print.o | egrep "hello$|print$"
     9: 00000000     4 OBJECT  GLOBAL DEFAULT    3 hello
    10: 00000000    23 FUNC    GLOBAL DEFAULT    1 print
```

`LOCAL` 和 `GLOBAL` 直观地反应了 static 用于限定变量和函数在文件之外是否可访问。加了 static 以后，文件之外不可见。

补充另外一个 nm 工具的结果，针对加 `static` 的情况：

```
$ nm print.o | egrep "hello$|print$"
00000000 d hello
00000000 t print
```

不加的情况：

```
$ nm print.o | egrep "hello$|print$"
00000000 D hello
00000000 T print
```

上面四个字母有两组大小写，分别对应 data, text 的 LOCAL 和 GOLOBAL 符号，其中 "hello" 是数据，“print” 作为函数处在代码区域。

> man nm:
>
> "D" "d" The symbol is in the initialized data section.
>
> "T" "t" The symbol is in the text (code) section.
>
> If lowercase, the symbol is usually local; if uppercase, the symbol is global (external). There are however a few lowercase symbols that are shown for special global symbols ("u", "v" and "w")

延伸介绍到 nm 这个工具是因为，Linux 内核的 System.map 这样的符号表文件经常会被用来调试，这个文件实际上是用 nm 导出来的。

再延伸一个 WEAK 类型，这个类型类似于不加 static 的 GLOBAL，但是呢，允许定义另外一个同名的函数或者变量，用来覆盖 WEAK 类型的这个：

```
$ cat print.c
#include <stdio.h>

__attribute__((weak)) char *hello = "hello";

__attribute__((weak)) void print(char *str)
{
  printf("%s\n", str);
}

$ cat hello.c
#include "print.h"

char *hello = "hello, world";

int main(void)
{
  print(hello);
  return 0;
}

$ ./hello
hello, world
```

这种情况允许某个变量或者函数的“multiple definition”，如果不定义为 WEAK 类型而且不定义为 LOCAL（用 static），这种情况本来是不被允许的：

```
$ gcc -m32 -o hello x.c print.h print.c
/tmp/ccMO5y0A.o:(.data+0x0): multiple definition of `hello'
/tmp/ccj8KK1s.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

这种用法在内核中被广泛采用，通常用来确保可以添加架构特定的优化函数：

```
$ grep __weak -ur ./ --include "*.c" | wc -l
413
```

汇总如下：

关键字	类型	说明
static	LOCAL	限文件内访问
不加 static	GLOBAL	文件外可在 extern 声明后访问
weak attribute	WEAK	同 GLOBAL，但可重定义

从运行的角度

上面从编译和二进制程序文件的角度分析了 `static` 关键字针对文件层面变量和函数的约定，下面再来看看函数内部的变量，在声明为 `static` 与否情况下的异同。

作为对比，把其他类型的变量也纳入进来：

```
$ cat hello.c
#include <stdio.h>

static int m;
static int n = 1000;
int a;
int b = 10000;

static int hello(void)
{
    static int i;
    static int j = 10;
    int x;
    int y = 100;
    register int z = 33;
```

```
    printf("i = %d, addr of i = %p\n", i, &i);
    printf("j = %d, addr of j = %p\n", j, &j);
    printf("x = %d, addr of x = %p\n", x, &x);
    printf("y = %d, addr of y = %p\n", y, &y);
    printf("z = %d, in register, no addr\n", z);

    return 0;
}

int main(int argc, char *argv[])
{
    printf("argc = %d, addr of argc = %p\n", argc, &argc);
    printf("argv = %s, addr of argv = %p\n", argv[0], argv);
    printf("m = %d, addr of m = %p\n", m, &m);
    printf("n = %d, addr of n = %p\n", n, &n);
    printf("a = %d, addr of a = %p\n", a, &a);
    printf("b = %d, addr of b = %p\n", b, &b);

    hello();

    return 0;
}

$ gcc -m32 -o hello hello.c
$ ./hello
argc = 1, addr of argc = 0xffd91f60
argv = ./hello, addr of argv = 0xffd91ff4
m = 0, addr of m = 0x804a030
n = 1000, addr of n = 0x804a020
a = 0, addr of a = 0x804a038
b = 10000, addr of b = 0x804a024
i = 0, addr of i = 0x804a034
j = 10, addr of j = 0x804a028
x = -143124200, addr of x = 0xffd91f24
y = 100, addr of y = 0xffd91f28
z = 33, in register, no addr
```

用二进制程序文件来佐证：

```
$ readelf -S hello | grep 804a | tail -2
  [24] .data PROGBITS 0804a018 001018 000014 00  WA 0 0 4
  [25] .bss  NOBITS   0804a02c 00102c 000010 00  WA 0 0 4

$ readelf -s hello |  egrep " m$| n$| a$| b$| i| j| x$| y$"
    36: 0804a030     4 OBJECT  LOCAL  DEFAULT   25 m
    37: 0804a020     4 OBJECT  LOCAL  DEFAULT   24 n
    39: 0804a034     4 OBJECT  LOCAL  DEFAULT   25 i.2021
    40: 0804a028     4 OBJECT  LOCAL  DEFAULT   24 j.2022
    54: 0804a024     4 OBJECT  GLOBAL DEFAULT   24 b
    67: 0804a038     4 OBJECT  GLOBAL DEFAULT   25 a
```

综合上面 3 组数据：

类型	代码	存储区域
文件内	static int m;	.bss（被初始化为 0）
文件内	static int n=1000;	.data
文件内	int a;	.bss（被初始化为 0）, GLOBAL
文件内	int b = 10000;	.data, GLOBAL
函数内	static int i;	.bss（被初始化为 0）
函数内	static int j = 10;	.data
函数内	int x;	stack（值随机，未初始化）
函数内	int y = 100;	stack
函数内	register int z = 33;	register（汇编中分配好）
函数参数	argc, argv	stack（默认调用约定）

再补充几点：

1. 用 register 定义的变量存放在寄存器中，所以无法获取它们的内存地址（因为根本不存放在内存中）。可以通过查看汇编代码确认：

```
$ gcc -m32 -S -o hello.s hello.c
$ grep 33 hello.s
  movl  $33, %ebx
```

2. 函数内用 static 定义的变量名（i 和 j）在符号表中都加了后缀，主要是方便多个函数定义同样的变量名，因为这些变量仅限该函数内（含多次调用）可见。

3. 函数内非 static 定义的变量，以及函数参数的传递都是通过 Stack 完成的，这些变量只在函数内（包括 Caller, Callee）可见，外部不可见，所以在符号表中也找不到它们。

4. 关于函数参数传递，如果明确改变了调用约定，比如函数明确加了 `\_\_attribute\_\_((fastcall))` 声明，那么部分参数将通过寄存器传递。不过 main 是例外，因为它的 Caller（ `__libc_start_main` ）默认是通过 Stack 传递参数的，再改变它的调用约定就拿不到正确的数据了。

小结

大学的课程蛮多都停留在语法层面的描述，需要透彻理解一些“概念”，还是需要结合实际操作，从终极使用的角度来看这些“概念”，看到的深度和细节会大有不同。

如果想与本文作者更深入地探讨程序链接、装载和运行原理，欢迎订阅吴老师的 10 小时 C 语言进阶视频课：《360° 剖析 Linux ELF》。

转发本文后截图发给 tinylab，赠送该系列 PDF 合集。

> by Falcon of TinyLab.org 2015/07/29

笔者最早在 2008 年写过一篇《如何为可执行文件“减肥”》。

该文通过各种努力，把可打印 `Hello World` 的 Linux ELF 可执行文件，从 6442 个字节，减少到 76 字节。

并且早期也有在知乎的相关问题下回复：Windows 上最小的“HelloWorld.exe”能有多小？，不过反响较小。

最近笔者抽空整理该文，并在重读 A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux（史上最小 Linux ELF 文件，即 ELF Kickers 作者，不过该 ELF 只能产生一个返回值）的基础上，做了进一步突破，先后造出了 52 字节和 45 字节的 ELF 可执行文件，并且可以打印 `Hello World`。

而社区的早期纪录是 59 字节，也是 ELF Kickers 作者创造的，文件在 ELF Kickers 的 `tiny/hello`：

```
$ git clone https://github.com/BR903/ELFkickers
$ cd ELFkickers
$ wc -c tiny/hello
59 tiny/hello
```

其中 52 字节的突破主要借鉴 A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux 关于把 ELF 程序头完全合并进 ELF 文件头的努力，而 45 字节的突破除了继承自己早期那篇文章的用参数赋值的想法外，还有些幸运的因素在（通过非法系统调用可以正常退出程序）。

最后，不仅获得了一个二进制只有 10 字节的 `hello.s`：

```
    .text
    .global _start      # stack differ from main()
_start:                 # stack: argc, argv[0], argv[1],
argv[2]
    pop    %edx         # argc, number of argv[]
                        # 3rd arg: string length: $len
    pop    %ecx         # argv[0], program name with path
                        # 2rd arg: string to write

    #xor   %ebx, %ebx   # 1st arg: output: stdout:1, use
stdin:0
                        # 1st arg: exit code for sys_exit
    mov     $4, %al     # (1) write our string to stdout
                        # sys_write: syscall number = 4
    int    $0x80        # call kernel
    mov     $1, %al     # (2) and exit
                        # sys_exit:  syscall number = 1
    int    $0x80        # call kernel
```

最后，完全把 ELF 程序头和代码合并进了 ELF 文件头，而且可以打印字符串。

而最重要的是，通过这个过程，可以透彻地理解 C 语言的开发过程，关于 C、汇编、操作系统背后的很多奥秘。

由于最近把早期的一些文章整理成了《C 语言编程透视》一书，相关记录也更新到了该书。

如果想了解该 ELF 裁剪的细节，请移步《C 语言编程透视》GitBook 版，并阅读第 10 章。

欢迎做进一步交流，关注我们的微博@泰晓科技 或者关注我们的开源书籍项目 Github 源码仓库。也欢迎提交 Pull Request。

21:43:17

> By Falcon of TinyLab.org Dec 02, 2019

背景简介

作为一门特别重视实践的课程，《360° 剖析 Linux ELF》 视频课程自 1 个月前上线以来，一直在持续迭代和更新实验材料。

- 今日新增了 15 种代码执行的方式，包括 exec, shlib, dlopen, cscript, binfmt-misc, embed, load-malloc, load-mmap 等
- 相比视频上线时，已新增接近 30 份实验材料，至今累计提供了 70 多份实验材料
- 课程上线以后，持续连载了多篇 Linux ELF 系列文章，这些文章基本都是综合型的实验例子

本文将简要介绍这 15 种代码执行的方式，随后，展示一下该课程实验材料的当前数据统计结果。

15 种代码执行的方式

在第 7 章，该课程详细介绍了代码运行与退出，为了更透彻地理解这部分内容，经过精心的准备，我们新增了如下 15 份实验材料，分别介绍了 15 种代码执行的方式：

实验材料	材料说明
exec	编译成可执行文件
shlib	编译成共享库，并直接执行共享库
dlopen	通过 dlopen 直接解析代码符号并运行
cscript	以脚本的方式运行 C 语言程序
emulator	通过模拟器运行可执行文件
binfmtc-extension	通过扩展名直接运行 C 语言程序
binfmtc-magic	自定义一个魔数直接运行 C 语言程序
embed-ldscript	嵌入到另一个程序运行，用 ldscript 辅助
embed-objcopy	嵌入到另一个程序运行，用 objcopy 辅助
embed-pic	嵌入到另一个程序运行，代码本身实现位置无关
load-malloc-auto	通过程序加载到 malloc 分配的内存后运行，运行时确定数据地址
load-malloc-pic	通过程序加载到 malloc 分配的内存后运行，使用位置无关代码
load-mmap-auto	通过程序加载到 mmap 申请的内存后运行，运行时确定数据地址
load-mmap-fixed	通过程序加载到 mmap 申请的内存后运行，使用固定地址
load-mmap-pic	通过程序加载到 mmap 申请的内存后运行，使用位置无关代码

每一份实验材料都有配套的 Makefile 和相应的 C 或者汇编语言代码。这些材料完整地呈现了 Linux 程序执行的奥秘。

其他数据统计

该视频课程除了提供 8 份演示幻灯（含数十张全新设计的图表），10 小时视频以外，其实验材料多达 70 多份，并且还在持续迭代和更新中。

截止到现在，相应的数据统计如下：

标准文档多达 **25** 份

```
$ find ./ -name "*.pdf" | wc -l
25
```

使用说明多达 **53** 份

```
$ find ./ -name "*.md" | wc -l
53
```

另有 **13** 份演示小视频

```
$ grep showterm.io -ur ./0* | wc -l
13
```

多达 **5351** 行实验代码

通过 cloc 统计后，得到如下更为详细的数据，内含 55 个 C 程序和 23 份汇编代码，另有 50 个 Makefile。

```
$ cloc ./
     287 text files.
     264 unique files.
     605 files ignored.

-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               55            451             166           1710
make                            50            474              76           1329
Assembly                        23            152              62           1044
C++                              3            127              13            971
CMake                            9             51              25            286
C/C++ Header                     2              2               1              6
Bourne Shell                     1              2               0              5
-------------------------------------------------------------------------------
SUM:                           143           1259             465           5351
-------------------------------------------------------------------------------
```

详细目录结构如下

```
$ tree -L 3 -d ./
```

```
./
├── 01-init
│   ├── hello-asm
│   ├── hello-c
│   ├── linux-lab
│   └── vim
├── 02-born
│   ├── run-a.out
│   ├── run-arm
│   │   └── arm
│   ├── run-elf
│   ├── run-macho
│   └── run-pe
├── 03-face
│   ├── elf-tools
│   ├── gcc-usage
│   ├── tcc-usage
│   ├── tiny-executables
│   │   ├── elf
│   │   ├── mach-o
│   │   └── pe
│   └── upx
├── 04-basic
│   ├── addr-align
│   │   └── build
│   ├── big-little-endian
│   ├── c-std
│   ├── debug-example
│   ├── elf-objects
│   ├── exec-shlib
│   ├── inline-asm
│   ├── sections
│   └── shlib-pic
├── 05-generate
│   ├── assembly
│   ├── dlopen
│   ├── gcc-compile-background
```

```
│   ├── mini-dynamic-linker
│   └── standalone-c
├── 06-execute
│   ├── args-and-env
│   ├── assembly
│   │   └── staged
│   ├── fork-and-exec
│   ├── myinit
│   └── tinyshell
├── 07-running
│   ├── assembly -> ../06-execute/assembly/
│   ├── binfmtc
│   ├── c-variables
│   ├── mem-layout
│   ├── monitor-myself
│   ├── ptrace
│   ├── run-code
│   │   ├── binfmtc-extension
│   │   ├── binfmtc-magic
│   │   ├── cscript
│   │   ├── dlopen
│   │   ├── embed-ldscript
│   │   ├── embed-objcopy
│   │   ├── embed-pic
│   │   ├── emulator
│   │   ├── exec
│   │   ├── load-malloc-auto
│   │   ├── load-malloc-pic
│   │   ├── load-mmap-auto
│   │   ├── load-mmap-fixed
│   │   ├── load-mmap-pic
│   │   └── shlib
│   ├── tcc-run
│   └── weak
└── standards
    └── svid
```

```
72 directories
```

小结

上述 15 份实验材料，随同其他资料都已经上传到代码仓库，该代码仓库目前只面向学员开放。

订阅该课程：《360° 剖析 Linux ELF》，可即刻下载上述实验材料，并同吴老师以及数百学员一起研究和讨论 Linux 程序的链接、装载和运行奥秘，提升分析和解决实际问题的效率。

所有上述实验材料，都可以直接在 Linux Lab 下即时进行实验。而 Linux Lab 正常情况下只需要几十分钟就可以安装完毕，它不仅支持 Linux，还可以直接在新版的 Windows 和 MacOSX 下运行，只要有 Docker 环境即可。

转发本文后截图发给 tinylab，赠送 Linux ELF 系列文章 PDF 合集。

该文件修订时间： 2021-01-19 21:43:17

> By Falcon of TinyLab.org Dec 09, 2019

背景简介

有一天，某位同学在讨论群聊起来：

> 除了直接把 C 语言程序编译成 ELF 运行以外，是否可以转成二进制，然后通过第三方程序加载到内存后再运行。

带着这样的问题，我们写了四篇文章，这是其一。

这一篇先介绍如何把 ELF 文件转为二进制文件，然后把二进制文件作为一个 Section 加入到另外一个程序，然后在那个程序中访问该 Section 并运行。

准备工作

先准备一个非常简单的 X86 汇编：

```
# hello.s
# as --32 -o hello.o hello.s
# ld -melf_i386 -o hello hello.o
# objcopy -O binary hello hello.bin
#

    .text
.global _start
_start:
    xorl   %eax, %eax
    movb   $4, %al                  # eax = 4, sys_write(fd, ad
dr, len)
    xorl   %ebx, %ebx
    incl   %ebx                     # ebx = 1, standard output
    movl   $.LC0, %ecx              # ecx = $.LC0, the addr of
string
    xorl   %edx, %edx
    movb   $13, %dl                 # edx = 13, the length of .
string
    int    $0x80
    xorl   %eax, %eax
    movl   %eax, %ebx               # ebx = 0
    incl   %eax                     # eax = 1, sys_exit
    int    $0x80

    .section .rodata
.LC0:
    .string "Hello World\xa\x0"
```

这段代码编译、运行后可以打印 Hello World。

通过 **objcopy** 转换为二进制文件

先来转换为二进制文件，可以用 objcopy。

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ objcopy -O binary hello hello.bin
```

分析转换过后的二进制代码和数据

如果要用 objcopy 做成 Binary 还能运行，怎么办呢?

首先来分析一下，转成 Binary 后的代码和数据如下：

```
$ hexdump -C hello.bin
00000000  31 c0 b0 04 31 db 43 b9  6d 80 04 08 31 d2 b2 0d  |1.
..1.C.m...1...|
00000010  cd 80 31 c0 89 c3 40 cd  80 48 65 6c 6c 6f 20 57  |..
1...@..Hello W|
00000020  6f 72 6c 64 0a 00 00
```

可以发现，刚好只保留了代码和数据部分，其他控制相关的内容全部不见了，非常“纯正”。

再用 objdump 对照看看：

```
$ objdump -d -j .text hello

hello1:     file format elf32-i386


Disassembly of section .text:

08048054 <_start>:
 8048054:    31 c0                   xor    %eax,%eax
 8048056:    b0 04                   mov    $0x4,%al
 8048058:    31 db                   xor    %ebx,%ebx
 804805a:    43                      inc    %ebx
 804805b:    b9 6d 80 04 08          mov    $0x804806d,%ecx
 8048060:    31 d2                   xor    %edx,%edx
 8048062:    b2 0d                   mov    $0xd,%dl
 8048064:    cd 80                   int    $0x80
 8048066:    31 c0                   xor    %eax,%eax
 8048068:    89 c3                   mov    %eax,%ebx
 804806a:    40                      inc    %eax
 804806b:    cd 80                   int    $0x80
```

需要注意这一行：

```
 804805b:    b9 6d 80 04 08          mov    $0x804806d,%ecx
```

在 Binary 中，数据地址是被写死的。

所以，要让 hello.bin 能够运行，必须要把这段 Binary 装载在指定的位置，即：

```
$ nm hello | grep " _start"
08048054 T _start
```

这样取到的数据位置才是正确的。

如何运行转换过后的二进制

这个是内核压缩支持的惯用做法，先要取到 Load Address，告诉 wrapper kernel，必须把数据解压到 Load Address 开始的位置。

如果这个要在用户空间做呢?

看上去没那么容易，不过可以这么做：

1. 把 hello.bin 作为一个 Section 加入到目标执行代码中，比如叫 run-bin.c
2. 然后写 ld script 明确把 hello.bin 放到 Load Address 地址上
3. 同时需要修改 ld script 中 run-bin 本身的默认加载地址，否则就覆盖了。也可以先把 hello 的 Load Address 往后搬动，这里用前者。

具体实现

先准备一个 run-bin.c，"bin_entry" 为 hello.bin 的入口，通过 ld script 定义。

```
#include <stdio.h>

extern void bin_entry(void);

int main(int argc, char *argv[])
{
    bin_entry();
    return 0;
}
```

接着，先拿到 run-bin.o：

```
$ gcc -c -o run-bin.o run-bin.c
```

把 hello.bin 作为 `.bin` section 加入进 run-bin.o：

```
$ objcopy --add-section .bin=hello.bin --set-section-flags .bin
=contents,alloc,load,readonly run-bin.o
```

最后，修改链接脚本，先拿到一个默认的链接脚本作为 base：

```
$ ld -melf_i386 --verbose | sed -ne "/=======/,/=======/p" | gr
ep -v "=======" > ld.script
```

修改如下，得到一个新的 ld.script.new：

```
$ git diff ld.script ld.script.new
diff --git a/ld.script b/ld.script.new
index 91f8c5c..7aecbbe 100644
--- a/ld.script
+++ b/ld.script.new
@@ -11,7 +11,7 @@ SEARCH_DIR("=/usr/local/lib/i386-linux-gnu");
 SEARCH_DIR("=/lib/i386-linux-gnu")
 SECTIONS
 {
   /* Read-only sections, merged into text segment: */
-  PROVIDE (__executable_start = SEGMENT_START("text-segment",
0x08048000)); . = SEGMENT_START("text-segment", 0x08048000) + S
IZEOF_HEADERS;
+  PROVIDE (__executable_start = SEGMENT_START("text-segment",
0x08046000)); . = SEGMENT_START("text-segment", 0x08046000) + S
IZEOF_HEADERS;
   .interp         : { *(.interp) }
   .note.gnu.build-id : { *(.note.gnu.build-id) }
   .hash           : { *(.hash) }
@@ -60,6 +60,11 @@ SECTIONS
     /* .gnu.warning sections are handled specially by elf32.em
.  */
     *(.gnu.warning)
   }
+  .bin 0x08048054:
+  {
+    bin_entry = .;
+    *(.bin)
+  }
   .fini           :
   {
     KEEP (*(SORT_NONE(.fini)))
```

主要做了两笔修改：

1. 把 run-bin 的执行地址往前移动到了 0x08046000，避免代码覆盖

2. 获取到 hello 的 _start 入口地址，并把 .bin 链接到这里，可以通过 `nm hello | grep " _start"` 获取

```
$ nm hello | grep " _start"
08048054 T _start
```

3. 把 bin_entry 指向 .bin section 链接后的入口

最后，用新的链接脚本链接如下：

```
$ gcc -m32 -o run-bin run-bin.o -T ld.script.new
```

链接后可以完美运行：

```
$ ./run-bin
Hello World
```

小结

在这个基础上，加上压缩/解压支持，就可以类似实现前面文章中提到的 UPX 了，即类似内核压缩/解压支持。

本文的方法不是很灵活，要求必须把 Binary 装载在指定的位置，否则无法正确获取到数据，后面我们继续讨论如何消除这种限制。

另外，本文用到了 ld script 来设定程序和 Section 的加载地址，实际上可以完全通过 objcopy 和 gcc 的参数来指定，从而减少不必要的麻烦。具体用法欢迎订阅吴老师的 10 小时 C 语言进阶视频课：《360° 剖析 Linux ELF》，课程提供了超过 70 多份实验材料，其中 15 个例子演示了 15 种程序执行的方法。

> By Falcon of TinyLab.org Dec 09, 2019

背景简介

有一天，某位同学在讨论群聊起来：

> 除了直接把 C 语言程序编译成 ELF 运行以外，是否可以转成二进制，然后通过第三方程序加载到内存后再运行。

带着这样的问题，我们写了四篇文章，这是其二。

上篇 介绍了如何把 ELF 文件转成二进制文件，并作为一个新的 Section 加入到另外一个程序中执行。

这个代码包括两个段，一个 text 段，一个 data 段，默认链接完以后，text 中是通过绝对地址访问 data 的，ELF 转成 Binary 后，这个地址也写死在 ELF 中，如果要作为新的 Seciton 加入到另外一个程序，那么链接时必须确保 Binary 文件的加载地址跟之前的 ELF 加载地址一致，否则数据存放的位置就偏移了，访问不到，所以上篇文章用了一个客制化的 ld script，在里头把 Binary Seciton 的加载地址（运行时地址）写死的。

让数据地址与加载地址无关

本篇来讨论一个有意思的话题，那就是，是否可以把这个绝对地址给去掉，只要把这个 Binary 插入到新程序的 Text 中，不关心加载地址，也能运行?

想法是这样：data 应该跟 text 关联起来，也就是说，用相对 .text 的地址，因为 Binary 里头的 .rodata 是跟在 .text 后面，在文件中的相对位置其实是固定的，是否可以在运行时用一个偏移来访问呢？也就是在运行过程中，获取到 .text 中的某个位置，然后通过距离来访问这个数据?

在运行时获取 **eip**

由于加载地址是任意的，用 .text 中的符号也不行，因为在链接时也一样是写死的

（用动态链接又把问题复杂度提升了），所以，唯一可能的办法是 eip，即程序地址计数器。

但是 eip 是没有办法直接通过寄存器获取的，得通过一定技巧来，下面这个函数就可以：

```
eip2ecx:
    movl   (%esp), %ecx
    ret
```

这个函数能够把 eip 放到 ecx 中。

原理很简单，那就是调用它的 call 指令会把 next eip 放到 stack，并跳到 eip2ecx。所以 stack 顶部就是 eip。这里也可以直接用 `pop %ecx` 。

所以这条指令能够拿到 `.here` 的地址，并且存放在 `ecx` 中：

```
    call   eip2ecx
.here:
    ...

    .section .rodata
.LC0:
    .string "Hello World\xa\x0"
```

通过 **eip** 与数据偏移计算数据地址

然后接下来，由于汇编器能够算出 .here 离 .LC0（数据段起始位置）：`.LC0 - .here` ，对汇编器而言，这个差值就是一个立即数。如果在 `ecx` 上加上（addl）这个差值，是不是就是数据在运行时的位置?

我们在 .here 放上下面这条指令：

```
  call   eip2ecx
.here:
    addl   $(.LC0 - .here), %ecx
    ...

    .section .rodata
.LC0:
    .string "Hello World\xa\x0"
```

同样能够拿到数据的地址，等同于：

```
movl   $.LC0, %ecx              # ecx = $.LC0, the addr of stri
ng
```

下面几个综合一起回顾：

- `addl` 这条指令的位置正好是运行时的 next eip （call 指令的下一条）
- `.here` 在汇编时确定，指向 next eip
- `.LC0` 也是汇编时确定，指向数据开始位置
- `.LC0 - .here` 刚好是 `addl` 这条指令跟数据段的距离/差值
- `call eip2ecx` 返回以后，ecx 中存了 eip
- `addl` 这条指令把 ecx 加上差值，刚好让 ecx 指向了数据在内存中的位置

完整代码如下：

```
# hello.s
#
# as --32 -o hello.o hello.s
# ld -melf_i386 -o hello hello.o
# objcopy -O binary hello hello.bin
#

    .text
.global _start
_start:
```

```
    xorl   %eax, %eax
    movb   $4, %al                  # eax = 4, sys_write(fd, ad
dr, len)
    xorl   %ebx, %ebx
    incl   %ebx                     # ebx = 1, standard output

    call   eip2ecx
.here:
    addl   $(.LC0 - .here), %ecx    # ecx = $.LC0, the addr of
string
                                    # equals to: movl   $.LC0,
%ecx

    xorl   %edx, %edx
    movb   $13, %dl                 # edx = 13, the length of .
string
    int    $0x80
    xorl   %eax, %eax
    movl   %eax, %ebx               # ebx = 0
    incl   %eax                     # eax = 1, sys_exit
    int    $0x80

eip2ecx:
    movl   (%esp), %ecx
    ret

    .section .rodata
.LC0:
    .string "Hello World\xa\x0"
```

链接脚本简化

这个生成的 hello.bin 链接到 run-bin，就不需要写死加载地址了，随便放，而且不需要调整 run-bin 本身的加载地址，所以 ld.script 的改动可以非常简单：

```
$ git diff ld.script ld.script.new
diff --git a/ld.script b/ld.script.new
index 91f8c5c..e14b586 100644
--- a/ld.script
+++ b/ld.script.new
@@ -60,6 +60,11 @@ SECTIONS
     /* .gnu.warning sections are handled specially by elf32.em
.  */
     *(.gnu.warning)
   }
+  .bin           :
+  {
+    bin_entry = .;
+    *(.bin)
+  }
   .fini           :
   {
     KEEP (*(SORT_NONE(.fini)))
```

直接用内联汇编嵌入二进制文件

在这个基础上，可以做一个简化，直接用 `.pushsection` 和 `.incbin` 指令把 `hello.bin` 插入到 run-bin 即可，无需额外修改链接脚本：

```
$ cat run-bin.c
#include <stdio.h>

asm (".pushsection .text, \"ax\" \n"
     ".globl bin_entry \n"
     "bin_entry: \n"
     ".incbin \"./hello.bin\" \n"
     ".popsection"
);

extern void bin_entry(void);

int main(int argc, char *argv[])
{
    bin_entry();
    return 0;
}
```

这个内联汇编的效果跟上面的链接脚本完全等价。

把数据直接嵌入代码中

进一步简化汇编代码把 eip2ecx 函数去掉:

```
# hello.s
#
# as --32 -o hello.o hello.s
# ld -melf_i386 -o hello hello.o
# objcopy -O binary hello hello.bin
#

    .text
.global _start
_start:
    xorl   %eax, %eax
    movb   $4, %al                  # eax = 4, sys_write(fd, ad
dr, len)
    xorl   %ebx, %ebx
    incl   %ebx                     # ebx = 1, standard output

    call   eip2ecx
eip2ecx:
    pop    %ecx
    addl   $(.LC0 - eip2ecx), %ecx  # ecx = $.LC0, the addr of
string
                                    # equals to: movl   $.LC0,
%ecx

    xorl   %edx, %edx
    movb   $13, %dl                 # edx = 13, the length of .
string
    int    $0x80
    xorl   %eax, %eax
    movl   %eax, %ebx               # ebx = 0
    incl   %eax                     # eax = 1, sys_exit
    int    $0x80

.LC0:
    .string "Hello World\xa\x0"
```

再进一步，直接把数据搬到 next eip 所在位置：

```
# hello.s
#
# as --32 -o hello.o hello.s
# ld -melf_i386 -o hello hello.o
# objcopy -O binary hello.o hello
#

    .text
.global _start
_start:
    xorl   %eax, %eax
    movb   $4, %al                  # eax = 4, sys_write(fd, ad
dr, len)
    xorl   %ebx, %ebx
    incl   %ebx                     # ebx = 1, standard output
    call   next                     # push eip; jmp next
.LC0:
    .string "Hello World\xa\x0"
next:
    pop    %ecx                     # ecx = $.LC0, the addr of
string
                                    # eip is just the addr of s
tring, `call` helped us
    xorl   %edx, %edx
    movb   $13, %dl                 # edx = 13, the length of .
string
    int    $0x80
    xorl   %eax, %eax
    movl   %eax, %ebx               # ebx = 0
    incl   %eax                     # eax = 1, sys_exit
    int    $0x80
```

小结

本文通过 eip + 偏移地址 实现了运行时计算数据地址，不再需要把 Binary 文件装载到固定的位置。

另外，也讨论到了如何用 `.pushsection/.popsection` 替代 ld script 来添加新的 Section，还讨论了如何把数据直接嵌入到代码中。

更多用法欢迎订阅吴老师的 10 小时 C 语言进阶视频课：《360° 剖析 Linux ELF》，课程提供了超过 70 多份实验材料，其中 15 个例子演示了 15 种程序执行的方法。

> By Falcon of TinyLab.org Dec 09, 2019

背景简介

有一天，某位同学在讨论群聊起来：

> 除了直接把 C 语言程序编译成 ELF 运行以外，是否可以转成二进制，然后通过第三方程序加载到内存后再运行。

带着这样的问题，我们写了四篇文章，这是其三。

前面两篇分别讨论了如何把一个转成 Binary 的 ELF 作为一个新的 Section 加入到另外一个程序中执行：

- ELF转二进制: 用 objcopy 把 ELF 转成 Binary 并运行
- ELF转二进制：允许把 Binary 文件加载到任意位置

如何实现动态加载和运行

本文继续讨论，但是方向是，在运行时加载 Binary 并运行，支持前面两篇中的两种类型的 Binary：绝对数据地址、相对数据地址。

先加载数据位置无关的 **Binary** 文件

先来考虑最简单的相对数据地址，动态加载后，仅需考虑把加载的 Binary 所在内存范围设置可执行即可。

把文件加载到内存中并设置内存保护属性的最佳方式是 mmap，当然也可以用 malloc/memalgin 分配内存然后用 mprotect 设置内存保护属性，但是需要额外考虑对齐。

可以直接基于 `man mmap` 中的例子小改一番，先拿到这个原始的例子：

```
$ man mmap | sed -ne "/Program source/,/SEE ALSO/p" | egrep -v
"Program|SEE" | sed -e "s/^       //g" > mmap.orig.c
```

做完如下修改，得到一个 mmap.new.c：

```
$ git diff mmap.orig.c mmap.new.c
diff --git a/mmap.orig.c b/mmap.new.c
index 640bcb0..fe039c8 100644
--- a/mmap.orig.c
+++ b/mmap.new.c
@@ -49,11 +49,12 @@ main(int argc, char *argv[])
          length = sb.st_size - offset;
      }

-    addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
+    addr = mmap(NULL, length + offset - pa_offset, PROT_READ|P
ROT_EXEC,
                  MAP_PRIVATE, fd, pa_offset);
      if (addr == MAP_FAILED)
          handle_error("mmap");

+#if 0
      s = write(STDOUT_FILENO, addr + offset - pa_offset, length
);
      if (s != length) {
          if (s == -1)
@@ -62,6 +63,9 @@ main(int argc, char *argv[])
          fprintf(stderr, "partial write");
          exit(EXIT_FAILURE);
      }
+#else
+    ((void (*)(void))addr)();
+#endif

      exit(EXIT_SUCCESS);
 }
```

上面的改动很简单，一方面是调整内存保护属性为 MAP_EXEC，另外一方面是把映射完的随机地址转换为一个 `void (*)void` 函数，然后直接执行，也就是调用

Binary，同时把之前的打印到控制台的部分注释掉。

以 -m32 参数编译，确保可以跑 -m32 的代码（继承上面两节）：

```
$ gcc -m32 -o mmap.new mmap.new.c
```

这个 mmap.new 即可运行第二篇得到的采用相对数据地址的 hello.bin：

```
$ ./mmap.new ./hello.bin 0
Hello World
```

再讨论数据位置固定的 **Binary** 文件

如果要带绝对地址的 hello.bin 呢?

由于数据加载地址是写死的，那意味着必须告知 mmap 映射到一个固定的地址，即 .text 的装载地址，否则数据访问就会出错。mmap 的第一个参数 addr 配合第三个参数 prot（设置为 MAP_FIXED），恰好可以做到。

只是，mmap 要求这个地址必须是对齐到页表的，这个 page size 可以通过 `sysconf(_SC_PAGE_SIZE)` 拿到。

可是，默认链接的时候，.text 段并不是对齐到 page size 的，对齐到 page size 的是 entry addr，还有一个 0x54 的偏移，即 elf header + program header。这个 0x54 末尾的地址不会是 page size 对齐的。

那意味者链接的时候得“做点手脚”，得强制让 .text 对齐到 page size，我们观察到 0x8046000 这个可以安全使用，因为程序都是从 0x8048000 之后的。当然，这里可用的只有不到 0x2000，8k，比这个大就把这个地址再改小吧。

怎么强制修改 .text 的装载地址呢，一个是上节提到的修改 ld.script，另外一个是直接用 ld 的 -Ttext 参数：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o -Ttext=0x8046000
```

之后，我们再做一些修改，允许传递这个地址给 mmap，得到 mmap.any.c：

```
$ diff --git a/mmap.orig.c b/mmap.any.c
index 640bcb0..aa23eb2 100644
--- a/mmap.orig.c
+++ b/mmap.any.c
@@ -11,7 +11,7 @@
 int
 main(int argc, char *argv[])
 {
-    char *addr;
+    char *addr = NULL;
     int fd;
     struct stat sb;
     off_t offset, pa_offset;
@@ -19,7 +19,7 @@ main(int argc, char *argv[])
     ssize_t s;

     if (argc < 3 || argc > 4) {
-        fprintf(stderr, "%s file offset [length]\n", argv[0]);
+        fprintf(stderr, "%s file offset [addr]\n", argv[0]);
         exit(EXIT_FAILURE);
     }

@@ -40,20 +40,16 @@ main(int argc, char *argv[])
     }

     if (argc == 4) {
-        length = atoi(argv[3]);
-        if (offset + length > sb.st_size)
-            length = sb.st_size - offset;
-                /* Can't display bytes past end of file */
-
-    } else {    /* No length arg ==> display to end of file */
-        length = sb.st_size - offset;
+        sscanf(argv[3], "%p", &addr);
     }
```

```
+    length = sb.st_size - offset;

-    addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
-                MAP_PRIVATE, fd, pa_offset);
+    addr = mmap((void *)addr, length + offset - pa_offset, PRO
T_READ|PROT_EXEC,
+                MAP_PRIVATE|MAP_FIXED, fd, pa_offset);
     if (addr == MAP_FAILED)
         handle_error("mmap");

+#if 0
     s = write(STDOUT_FILENO, addr + offset - pa_offset, length
);
     if (s != length) {
         if (s == -1)
@@ -62,6 +58,9 @@ main(int argc, char *argv[])
         fprintf(stderr, "partial write");
         exit(EXIT_FAILURE);
     }
+#else
+    ((void (*)(void))addr)();
+#endif

     exit(EXIT_SUCCESS);
 }
```

这个改动把原来的参数 length 换掉，替换为 addr，允许直接通过第三个程序参数设置 .text 的装载地址（确保数据地址有效）。

重新编译 mmap.any.c 并运行：

```
$ gcc -m32 -o mmap.any mmap.any.c
$ ./mmap.any ./hello.bin 0 0x8046000
Hello World
```

需要注意的是，这个地址必须与 -Ttext 指定的地址一致。

小结

到这里，ELF转二进制 3 篇文章就完成了，分别讨论了静态嵌入、位置无关和动态加载，接下来还有一篇会讨论，如何动态修改数据地址。

欢迎订阅吴老师的 10 小时 C 语言进阶视频课：《360° 剖析 Linux ELF》，课程提供了超过 70 多份实验材料，其中 15 个例子演示了 15 种程序执行的方法。

> By Falcon of TinyLab.org Dec 09, 2019

背景简介

有一天，某位同学在讨论群聊起来：

> 除了直接把 C 语言程序编译成 ELF 运行以外，是否可以转成二进制，然后通过第三方程序加载到内存后再运行。

带着这样的问题，我们写了四篇文章，这是其四。

前面几篇把汇编、静态链接、静态加载、动态加载都讲了，还差一个缺憾，那就是动态链接。

每次得加载到一个特定地址其实不是那么灵活，所以，在不修改代码的前提下，如果能保证加载到任意地址，而且还能访问数据的话，就需要“动态链接”。

动态计算数据地址

就是数据的地址可以动态分配，本文模拟一下这个过程。

这里用到的例子是最原始的 hello.s，未经 hack 的：

```
# hello.s
#
# as --32 -o hello.o hello.s
# ld -melf_i386 -o hello hello.o
# objcopy -O binary hello hello.bin
#

    .text
.global _start
_start:
    xorl   %eax, %eax
    movb   $4, %al                  # eax = 4, sys_write(fd, ad
dr, len)
    xorl   %ebx, %ebx
    incl   %ebx                     # ebx = 1, standard output
    movl   $.LC0, %ecx              # ecx = $.LC0, the addr of
string
    xorl   %edx, %edx
    movb   $13, %dl                 # edx = 13, the length of .
string
    int    $0x80
    xorl   %eax, %eax
    movl   %eax, %ebx               # ebx = 0
    incl   %eax                     # eax = 1, sys_exit
    int    $0x80

    .section .rodata
.LC0:
    .string "Hello World\xa\x0"
```

汇编后看看:

```
$ objdump -dr hello.o
hello.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:    31 c0                   xor    %eax,%eax
   2:    b0 04                   mov    $0x4,%al
   4:    31 db                   xor    %ebx,%ebx
   6:    43                      inc    %ebx
   7:    b9 00 00 00 00          mov    $0x0,%ecx
            8: R_386_32    .rodata
   c:    31 d2                   xor    %edx,%edx
   e:    b2 0d                   mov    $0xd,%dl
  10:    cd 80                   int    $0x80
  12:    31 c0                   xor    %eax,%eax
  14:    89 c3                   mov    %eax,%ebx
  16:    40                      inc    %eax
  17:    cd 80                   int    $0x80
```

我们希望能够动态链接，也就是在运行时根据 Binary 文件加载的位置，动态计算出数据地址（不用修改代码本身）。也就是说确认 `R_386_32` 这个类型的地址。

这个地址可以用 Binary 文件的加载地址加上 .rodata 相对 Binary 文件开头（即 .text 起始位置）的偏移算出来。

如上所示，.text 的大小是 0x19（0x17+2），那么 .rodata 跟 .text 的偏移刚好是 0x19，那数据地址可以通过 Binary load addr + 0x19 得出。其实这里也可以直接用 readelf -s hello.o 获得（看 Size 部分）：

```
$ readelf -S hello.o  | egrep " .text|Size"
  [Nr] Name              Type            Addr     Off    Size
 ES Flg Lk Inf Al
  [ 1] .text             PROGBITS        00000000 000034 000019
 00  AX  0   0  1
```

动态修改数据地址

另外一个是，这个地址需要编码回代码里头，即 "8: R_386_32 .rodata" 这里，也就是 .text 中的第 8 个字节开始的一个 32 位地址（the absolute 32-bit address of the symbol into the specified memory location，可参考：c - meaning of R_386_32/R_386_PC32 in .rel.text se...）

上面得到了两个关键数据：

1. 数据偏移（相对 .text 起始位置）：0x19
2. 代码偏移（相对 .text 其实位置）：8，这个地方要写入上面的地址

需要做的是，用 .text addr + 0x19 算出数据地址，然后写入 .text addr + 8 这个位置。

说明一下，在实际的动态链接中，上面两个地址都是可以根据 ELF 中的相关数据，动态计算出来的，这里因为是用到了 Binary 文件，我们通过 `objdump/readelf` 获取到数据访问指令地址和数据偏移后来模拟动态计算。

.text addr 这个地址在运行时确定，这里由 mmap 动态获得，接下来改动代码如下，核心部分如下：

```
/* dynamic linking: update the data address */
/* offset are gotten from this command: $ objdump -dr hello.o *
/
#define inst_offset 8
#define data_offset 0x19
*(unsigned int *)(addr + inst_offset) = (uint)addr + data_offse
t;
```

addr 即 mmap 加载 Binary 文件到内存时分配的地址，也是 .text 的起始地址（Binary 文件开头就是 .text，接着是 .rodata）。

完整代码如下：

```
$ diff --git a/mmap.orig.c b/mmap.any.c
index 640bcb0..87358d9 100644
```

```
--- a/mmap.orig.c
+++ b/mmap.any.c
@@ -11,7 +11,7 @@
 int
 main(int argc, char *argv[])
 {
-    char *addr;
+    char *addr = NULL;
     int fd;
     struct stat sb;
     off_t offset, pa_offset;
@@ -19,7 +19,7 @@ main(int argc, char *argv[])
     ssize_t s;

     if (argc < 3 || argc > 4) {
-        fprintf(stderr, "%s file offset [length]\n", argv[0]);
+        fprintf(stderr, "%s file offset [addr]\n", argv[0]);
         exit(EXIT_FAILURE);
     }

@@ -40,20 +40,25 @@ main(int argc, char *argv[])
     }

     if (argc == 4) {
-        length = atoi(argv[3]);
-        if (offset + length > sb.st_size)
-            length = sb.st_size - offset;
-                /* Can't display bytes past end of file */
-
-    } else {    /* No length arg ==> display to end of file */
-        length = sb.st_size - offset;
+        sscanf(argv[3], "%p", &addr);
     }
+    length = sb.st_size - offset;

-    addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
-                MAP_PRIVATE, fd, pa_offset);
```

```
+    addr = mmap((void *)addr, length + offset - pa_offset, PRO
T_READ|PROT_WRITE|PROT_EXEC,
+                MAP_PRIVATE|MAP_FIXED|MAP_POPULATE, fd, pa_off
set);
     if (addr == MAP_FAILED)
         handle_error("mmap");

+    /* dynamic linking: update the data address */
+    /* offset are gotten from this command: $ objdump -dr hell
o.o */
+    #define inst_offset 8
+    #define data_offset 0x19
+    *(unsigned int *)(addr + inst_offset) = (uint)addr + data_
offset;
+
+    /* flush icache: Using the GNU Compiler Collection (GCC):
Other Bui... */
+    __builtin___clear_cache(addr, addr + length + offset - pa_
offset);
+
+#if 0
     s = write(STDOUT_FILENO, addr + offset - pa_offset, length
);
     if (s != length) {
         if (s == -1)
@@ -62,6 +67,9 @@ main(int argc, char *argv[])
         fprintf(stderr, "partial write");
         exit(EXIT_FAILURE);
     }
+#else
+    ((void (*)(void))addr)();
+#endif

     exit(EXIT_SUCCESS);
 }
```

编译和运行如下：

```
$ gcc -m32 -o mmap.any mmap.any.c
$ sudo ./mmap.any ./hello.bin 0
Hello World
$ ./mmap.any ./hello.bin 0 0x8042000
Hello World
```

如上演示，这里不再需要提前把 hello 链接到一个“合法”的地址，而是让 mmap 自己合理分配一个，当然，也兼容自己指定一个合法的加载地址。

上述代码还加了一个函数，用于刷新指令 Cache，确保运行结果的一致性，这个函数是 gcc 提供的：`__builtin___clear_cache`，详细介绍在 c - Is there a way to flush the entire CPU cache...。

小结

未来可以考虑进一步完善 Binary 文件结构，在后面加入 inst offset 和 data offset 的数组，这样，在运行时就可以借用这个数据，不用在代码中写死。

到这里，四篇文章，比较系统地介绍了如何把一个 ELF 文件转换为 Binary 文件，并通过第三方程序加载和执行，内容涉及静态嵌入、位置无关、动态加载以及动态计算和修改数据地址。

欢迎订阅吴老师的 10 小时 C 语言进阶视频课：《360° 剖析 Linux ELF》，课程提供了超过 70 多份实验材料，其中 15 个例子演示了 15 种程序执行的方法。

> By Falcon of TinyLab.org Aug 20, 2019

课程序言

前段在知乎看到一个热门话题，有 1945 个关注者，898519 次阅读量。

这个话题是："从事嵌入式行业的你，现在年薪多少，有什么经历想和大家分享？"

这个话题的热度说明嵌入式的小伙伴们挺关心自己的职业发展和规划，那么，从事嵌入式行业（俗称底层开发）到底有没有前景呢？

答案是肯定的，以前的手机，现在的汽车、物联网、AIoT，没有一个离得开嵌入式，所以需求是很火爆的，既然有需求就有价值。那价值能到什么程度呢？当然取决于解决问题的实际能力。

曾经带过将近 100 人的团队，因为有人员流动，所以对整个行业的薪资待遇也比较清楚。据观察，骨干级别的优秀嵌入式软件工程师月薪能拿到 3 万左右，而独当一面的技术专家能拿到 4~5 万甚至更高。那如何才能做到独当一面呢？

一方面是打怪升级，就是丢到项目熔炉里头实战，另外一方面是，必须熟练掌握核心技术原理，建立起自己的知识体系，从而能够迅速、高效、甚至创造性地解决各类疑难问题。

下面这个课程，就是以程序链接、装载和运行作为切入点，以 ELF 剖析为主线，抽丝剥茧，通过大量具有挑战性的课程实验带领读者不断"打怪升级"，逐渐摸透各个原来似懂非懂的知识点，并且无缝地把它们串起来，从而做到以点带面，融会贯通，游刃有余。

接下来，就是一边得心应手地解决问题，一边踌躇满志地坐等 Leader 和 HR 沟通，准备升职加薪走上人生巅峰啦啦啦 ;-)

课程简介

近日，泰晓科技更新码农自我修养之「360° 剖析 Linux ELF」在线视频课程。

- 课程讲师

  吴章金，十年 Linux 研发经验，Linux Committer，前魅族内核团队技术总监。热门开源书《C 语言编程透视》作者。

- 开课时间

  课程视频将于 2019年10月19 日正式上线。

- 报名方式

  https://www.cctalk.com/m/group/88089283

阅读下文可进一步了解课程详情。

课程背景

关于 **“ELF 剖析”** 课程

作为嵌入式软件工程师，对于一些非常基础的知识点，有没有经常觉得似懂非懂，似是而非呢?

很多知识点，好像以前都学过，但是去分析和解决实际问题的时候，经常碰到：

- C 语言程序的入口是 main 函数，真的是这样吗?
- main 函数末尾必须加 `return 0` 吗，为什么?
- GCC 默认生成的 a.out 是 a.out 格式吗?
- Heap 与 Stack, Bss 与 Data 傻傻分不清！
- 碰到 Segmentation Fault, Stack Overflow 常常不知所措;
- 缓冲区注入攻击是什么东东，如何检测缓冲区溢出?
- 偶尔要用到 Linux AT&T 汇编语言或者内联汇编，怎么写，怎么学?
- 如果要让一个共享库可执行，可以做到吗?
- 如果无法获得共享库的源码，如何单独升级其中的库函数呢?
- 公司要做一个只有 8M 闪存的产品，如何下手?
- 要改变程序代码段的装载地址，该怎么做……

遇到这些问题不用灰心丧气，让我们一起来补补功课；那些曾经模棱两可的知识点，让我们以「实验」的方式都动手观察一遍，把那些模糊不清的地带摸清楚。

ELF 是 Linux 平台上最常用的一种可执行文件格式，它牵涉到程序的前世今生，所以通过剖析它，就可以把各个知识点串起来，通过工具切切实实观察程序编译和运行背后的机理，进而建立起完整的知识体系，做到知其然，也知其所以然。

该课程将以下面的问题发起：同样是打印一个 'hello' 字符串，gcc 8.3 默认生成的 a.out 有 8.3k，而裁剪到 45 字节也能实现同样功能，这 185 倍的差距是什么？

```
$ gcc --version | head -1
gcc (Ubuntu 8.3.0-16ubuntu3~16.04) 8.3.0
$ echo -ne '#include <stdio.h>\n int main(void) { printf("hello
\\n"); return 0; }' | gcc -x c - -
$ ./a.out
hello
$ wc -c a.out  # ~8.3k
8400

$ as --32 -o hello.o hello.s
$ ld -melf_i386 --oformat=binary -o hello hello.o
$ PATH=./:$PWD hello 0 0 0
hello
$ wc -c hello
45
```

本课程从 ELF 规范入手，从 X86 体系结构和指令集、X86 Linux AT&T 汇编，到代码预处理、编译、汇编、链接及 ELF 可执行文件的生成；再围绕 ELF 可执行文件的装载与运行，由浅入深地讲解程序加载、上下文传递、系统调用、内存布局、进程调度、代码调试以及程序退出全过程。在这个过程中，逐步讲解各知识点，并根据各知识点渐进地动手把一个 8.3k 的程序逐步瘦身到 45 个字节。通过动手实践，不仅可以掌握各种常用开发与调试工具，而且能够加深对各知识点的透彻理解。

本课程既可以顺序学习整体提升你对程序开发过程的理解，也可以针对具体问题学习具体章节。每章都有配套作业，可以在课程微信群讨论，多位 Linux 技术专家长期驻

群答疑。学习本课程，一方面可以解决自己工作中遇到的具体编程问题，另一方面可以打造自下而上的完整技能体系，在专业层级（Title & Salary）上有一个质的跃升。

本课程采用 Linux Lab 进行实验，只需一台支持 Docker 的 PC 或者笔记本，即可完成本课程所有实验。

指导老师

吴章金 / **Falcon**

十三年 Linux 系统使用经验，十年 Linux 内核研发经验，六年 Linux 团队管理经验。

重度开源践行者，Linux 官方社区贡献者，前魅族 Linux 内核团队（暨BSP部）技术总监。

- 2006 年参与创建兰大开源社区，2007 年负责组建正式的校园社团，并陆续支持直到 2010 年研究生毕业。
- 2009 年到龙芯梦兰实习，在两周内完成首版龙芯实时操作系统（Linux Preempt RT）的移植，同年，为 MIPS 平台开发了 Ftrace、内核压缩等支持，全面整理了龙芯 2 系所有设备的 Linux 内核，并陆续成功往官方社区提交。2010 年发起 Linux Loongson Community （LLC）项目，持续免费维护到 2012 年底，RMS 是 LLC 用户之一。2010 年被提名参加当年 Linux 内核峰会。
- 2010 年加入 WindRiver，从事 MIPS 平台 BSP 以及 Preempt RT, Ftrace 等 Features 研发。同年，筹备面向一线 Linux 工程师的公益性技术交流平台：泰晓科技。
- 2011 年加入魅族，从零开始组建系统优化团队，在业界率先思考体系化地解决系统稳定性、功耗、性能、发热等问题，协调研发、测试、生产、售后、流程管理等多个维度构建完整的系统核心体验保障体系。先后参与或者带领团队完成 20 多款手机系统底层软件的研发，总体规模达到数千万台。

在德累斯顿、珠海、北京和上海多次发表技术演讲，数年来，在泰晓科技公开发表了接近 200 篇技术文章，发起并维护了数个开源软件项目，撰写了多份论文和书籍，主要有:

- 2008/2015: 《C 语言编程透视》、《Shell 编程范例》
- 2009 RTLWS11: Porting RT-Preempt to Loongson2F
- 2011 RTLWS13: Tiny Linux Kernel Project: Section Garbage Collection Patchset
- 2013 ZhuhaiGDG: Android Linux 可靠性（RAS）研究与实践
- 2013 Packtpub: 《Instant Optimizing Embedded Systems Using BusyBox》
- 2015 CSDN MDCC 2015：智能手机系统优化的演进与实践
- 2016 CLK2016: Ftrace 实现原理与开发实践
- 2006~2019 开源项目
    - MIPS Ftrace
    - MIPS Preempt-RT
    - TinyLinux
    - Linux Lab
    - Linux 0.11 Lab
    - CS630 Qemu Lab
    - VnstatSVG
    - Linux Loongson Community
    - elinux.org 翻译等

欢迎阅读 细数我与 Linux 这十三年 进一步了解讲师。

这里 有讲师之前陆续发表的 200 多篇技术文章汇总。

内容大纲

1. 开篇：古有 “庖丁解牛”，今有 “码农剖 ELF”
    - 历史典故：庖丁解牛
        - 目无全牛：“三年之后，未尝见全牛也。……依乎天理，批大郤，导大窾，因其固然。”
        - 得心应手：“臣以神遇而不以目视，官知止而神欲行。”

        - 游刃有余：“以无厚入有间，恢恢乎其于游刃必有余地矣。”
        - 娴熟合律：“……莫不中音。合于桑林之舞，乃中经首之会。”
    - ELF 剖析：“Hello World!”
        - 8.3K v.s 45B
        - 185 倍差距背后隐藏了哪些不为人知的秘密
    - Linux Lab 实验环境准备
        - 简介、安装、演示
    - 实验作业
        - 下载并安装 Linux Lab 实验环境
        - 用 vim 编写 hello.c 并用 gcc 编译、运行
2. ELF 是什么?
    - Linux 支持哪些可执行文件格式
        - ELF：标准可执行文件格式，25 年历史
        - 跨系统运行 Windows、MacOSX 程序
        - X86 跨架构跑 ARM、MIPS、Risc-V 二进制程序
        - 其他可执行文件类型：Java、Python、Shell
    - 可执行文件格式的演进历史
        - a.out, coff, ELF 演进变迁
    - 各大操作系统都在用什么格式
        - Windows: PE
        - MacOSX: MacO
    - 实验作业
        - 在 Linux 下运行 Windows 程序
        - 在 X86 Linux 下运行 ARM 程序
        - 把上节编写的 hello.c 编译成 a.out 格式并与 ELF 比较
        - 进阶：在 Linux 下运行 MacOSX 程序
3. Hello 可执行文件可以有多小?
    - Linux 下默认编译的 Hello.c 有多大?
        - 初步分析 8.3K ELF 由哪几部分构成
    - 各大平台上的二进制 Hello 可执行文件做到了多小?
        - Windows PE：97 字节

        - MacOSX MacO：164 字节
        - Linux ELF：45 字节
    - 大家都做了哪些尝试?
        - 编译器优化
        - 手工编译和链接
        - 动手写汇编
        - 用系统调用取代库函数
        - 删掉不用的节区
        - 手工编辑二进制文件
        - 手工构造可执行文件
    - 实验作业
        - 为 hello.c 写一个 Makefile 并灵活调整 GCC 编译选项
        - 尝试运用您掌握的方法裁剪这个 hello 程序
        - 进阶：试试一个更快更小的 C 语言编译器 tinycc
4. 基础知识筹备
    - ELF 文件格式简介
        - 一起读 ELF 1.2 规范
        - Executables, Objects, Shared libraries, Core dumps
    - C 语言标准与函数调用规范
        - 标准：c89, c99, c11, gnu extensions
        - 函数调用规范：cdecl, fastcall, pascal
        - 静态检查：-Wall, -Werror, -Wpedantic
    - X86 体系结构知识
        - 对齐
        - 大小端
    - X86 AT&T 汇编语言
        - 指令集简介
        - 内联汇编怎么用
        - 自动生成第一个汇编语言程序
    - ELF 工具套装介绍
        - gcc, gdb, cpp, as, ld, ldd, ld.so
        - binutils: readelf, objdump, objcopy, strip

        - elfutils, ELF Kickers
        - hexdump, dd
        - gprof, gcov, perf
    - 实验作业
        - 用本节掌握的方法进一步裁剪 hello 程序
        - 生成一份干净可阅读的汇编代码
        - 用内联汇编读取某个寄存器
        - 写一个程序完成大小端转换
        - 查看某个程序用到了哪些共享库
        - 给定有多个错误的某程序，用上述工具编译、运行、反汇编、调试
        - 进阶：动手写一个 myreadelf
5. ELF 程序的生成
    - 代码编译与编译器优化
        - 编译过程揭秘：预处理、编译、汇编与链接
        - 编译器基础优化：-O2, -Os
        - 编译器进阶优化：gc-sections, branch-probabilities，lto
    - 静态链接和默认链接行为
        - GCC 默认链接参数详解
        - 探索 C 语言真正的入口
        - 默认链接脚本
    - 共享库和动态链接
        - 符号重定位：plt, got
        - 隐式使用 v.s 显式使用
        - preload: libs live patching
        - prelink: 事先链接取代运行时链接
    - 汇编语言版 Hello
        - 动手写第 1 个汇编语言程序：hello.s
    - 实验作业
        - 用本节掌握的方法进一步裁剪 hello 程序
        - 7 大架构 hello 汇编一块学
        - 无论输出到哪里，都允许 ls 输出带颜色
        - 编译一个内核，删除所有用不到的变量和函数

        - 修改内核 Makefile，禁止打开编译器优化
        - 进阶：动手写一个动态链接工具
6. ELF 程序的执行
    - 操作系统启动过程简介
        - 盘古开天辟地: 从电脑开机到第 1 个进程的诞生
        - 道生一，一生万物：pstree
        - 第 1 个可交互命令行程序的启动：Shell
    - 按键触发程序执行的那一刻
        - “程序” 有哪些类型
        - 键入一串字符并按下回车之后系统发生了什么?
        - 那些特殊的符号是如何解析的：`|, >, <, &`
        - env, ltrace, strace
    - 命令行参数和环境变量
        - 上下文传递
        - 用 gdb 观察
        - /proc/{comm, cmdline, environ}
    - 系统调用 Fork & Exec
        - 程序变进程的魔法
    - 实验作业
        - 用本节掌握的方法进一步裁剪 hello 程序
        - 女娲造人：写1个程序并作为 Linux 系统的第1个进程启动
        - 进阶：动手写一个简单的 Shell
7. ELF 程序的运行与退出
    - ELF 程序加载
        - 内核 `load_binary` 解读
    - 进程的内存布局
        - 内核 `do_mmap` 解读
        - 从程序内部、外部分析内存布局
    - 进程的运行和退出
        - 内核 `start_thread` 解读
        - 进程的运行和消亡

- 进程跟踪和调试原理
  - 用 gdb 跟踪和调试程序
  - 内核 `ptrace` 解读
- 实验作业
  - 用本节掌握的方法进一步裁剪 hello 程序
  - 写一个程序，在程序内部监控代码段是否被篡改
  - 在 498 行极小 OS 上装载、解析并运行标准 ELF 程序
  - 进阶：在 Linux 0.11 增加 ELF 支持
  - 进阶：设计一个可执行文件格式并添加 Linux 内核支持

8. 结语：像一个外科手术专家那样

- 把程序比作外科医生的病人，码农就是程序的医生
- 像专业的外科医生那样熟悉程序组织结构和运行机理
  - 熟练掌握 ELF 生成、执行和运行
  - 做到：目无全牛，游刃有余，“依乎天理……因其固然”
- 像敬业的外科医生那样给程序看病
  - 沉着冷静地 Debugging
  - 做到：得心应手，踌躇满志，“以神遇而不以目视，官知止而神欲行”

9. 参考书目一览

- 《庄子集解》
- 《C 语言编程透视》
- 《程序员的自我修养 —— 链接、装载与运行》
- 《Hacking Hello World》
- 《深入理解 Linux 内核》
- 《深入 Linux 内核架构》
- 《X86/X64 体系探索及编程》
- 《ARM 嵌入式系统开发 —— 软件设计与优化》
- 《MIPS 体系结构透视》
- 《Linux 内核完全注释 —— 基于 0.11 内核》

报名方式

报名地址：https://www.cctalk.com/m/group/88089283

直接扫码加泰晓科技主编微信：

该文件修订时间： 2021-01-19 21:43:17