Hello~, the smaller world!

在了解了 Hello 如何 World 之后，也许有同学会觉得可以对 Helloworld 做点什么。这里，让我们从可执行文件大小上入手，进一步加深对程序执行的理解。本文介绍如何将一个 Helloworld 可执行文件的大小从 11K 裁剪到 79 字节，当然也许你可以使它变得更小。：)

调试环境是 REDHAT9.0：

```
<onlyforos>[/home/onlyforos/hello]%gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix
--disable-checking --host=i386-redhat-linux --with-system-zlib
--enable-__cxa_atexit
Thread model: posix
gcc version 3.2 20020903 (Red Hat Linux 8.0 3.2-7)
<onlyforos>[/home/onlyforos/hello]%
```

首先看一下著名的 Hello,world 正常情况下的大小：

```
<onlyforos>[/home/onlyforos/hello]%cat hello.c
#include <stdio.h>
int main()
{
    printf("hello,world\n");
    return 0;
}

<onlyforos>[/home/onlyforos/hello]%gcc -o hello hello.c
<onlyforos>[/home/onlyforos/hello]%hello
hello,world
<onlyforos>[/home/onlyforos/hello]%ll hello
-rwxr-xr-x    1 onlyforos onlyforos    11362  6 月   9 15:15 hello*
<onlyforos>[/home/onlyforos/hello]%
```

11362 字节。这就是正常情况下 Hello,world 的大小。怎么样才能使它的尺寸变得更小一些呢？

经过《Hello~,ELF of the world!》中对 ELF 文件的分析，我们可以知道，各种符号信息在可执行文件中占有的很大比例。若想缩小可执行文件的体积，则首先我们就应该把这些符号信息去除。GCC 的-s 选项提供了和命令 strip 命令作用相同的功能，可以忽略来自输出文件的所有的符号信息。

```
<onlyforos>[/home/onlyforos/hello]%gcc -s -o hello1 hello.c
<onlyforos>[/home/onlyforos/hello]%ll hello1
-rwxr-xr-x    1 onlyforos onlyforos     2692  6 月   9 15:18 hello1*
```

我们看到，经过简单处理，hello 的体积已经大幅缩减。那么我们再用优化指令选项对其做进一步的压榨。通常优化选项使编译时间增加却可以使运行时间变短。

```
<onlyforos>[/home/onlyforos/hello]%gcc -s -O3 -o hello2 hello.c
```

```
<onlyforos>[/home/onlyforos/hello]%ll hello2
-rwxr-xr-x       1 onlyforos onlyforos        2672    6 月    9 15:18 hello2*
```

有 20 字节的效果，但看起来已经有些“ 技穷 ”了。2672 字节。我们来看看此时的可执行文件的内容：

```
<onlyforos>[/home/onlyforos/hello]%readelf –a hw1
ELF Header:
   Magic:      7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
   Class:                             ELF32
   Data:                              2's complement, little endian
   Version:                           1 (current)
   OS/ABI:                            UNIX - System V
   ABI Version:                       0
   Type:                              EXEC (Executable file)
   Machine:                           Intel 80386
   Version:                           0x1
   Entry point address:               0x8048274
   Start of program headers:          52 (bytes into file)
   Start of section headers:          1672 (bytes into file)
   Flags:                             0x0
   Size of this header:               52 (bytes)
   Size of program headers:           32 (bytes)
   Number of program headers:         6
   Size of section headers:           40 (bytes)
   Number of section headers:         25
   Section header string table index: 24

Section Headers:
   [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
   [ 0]                   NULL            00000000 000000 000000 00      0   0  0
   [ 1] .interp           PROGBITS        080480f4 0000f4 000013 00   A  0   0  1
   [ 2] .note.ABI-tag     NOTE            08048108 000108 000020 00   A  0   0  4
   [ 3] .hash             HASH            08048128 000128 000028 04   A  4   0  4
   [ 4] .dynsym           DYNSYM          08048150 000150 000050 10   A  5   1
4
   [ 5] .dynstr           STRTAB          080481a0 0001a0 00004a 00   A  0   0  1
   [ 6] .gnu.version      VERSYM          080481ea 0001ea 00000a 02   A  4   0  2
   [ 7] .gnu.version_r    VERNEED         080481f4 0001f4 000020 00   A  5   1  4
   [ 8] .rel.dyn          REL             08048214 000214 000008 08   A  4   0  4
   [ 9] .rel.plt          REL             0804821c 00021c 000010 08   A  4   b  4
   [10] .init             PROGBITS        0804822c 00022c 000018 00  AX  0   0  4
   [11] .plt              PROGBITS        08048244 000244 000030 04  AX  0   0  4
   [12] .text             PROGBITS        08048274 000274 0000f0 00  AX  0   0  4
   [13] .fini             PROGBITS        08048364 000364 00001c 00  AX  0   0  4
   [14] .rodata           PROGBITS        08048380 000380 000014 00   A  0   0  4
   [15] .data             PROGBITS        08049394 000394 00000c 00  WA  0   0  4
   [16] .eh_frame         PROGBITS        080493a0 0003a0 000004 00  WA  0   0  4
```

```
  [17] .dynamic          DYNAMIC         080493a4 0003a4 0000c8 08   WA  5   0
4
  [18] .ctors            PROGBITS        0804946c 00046c 000008 00   WA  0   0  4
  [19] .dtors            PROGBITS        08049474 000474 000008 00   WA  0   0  4
  [20] .jcr              PROGBITS        0804947c 00047c 000004 00   WA  0   0  4
  [21] .got              PROGBITS        08049480 000480 000018 04   WA  0   0  4
  [22] .bss              NOBITS          08049498 000498 000004 00   WA  0   0  4
  [23] .comment          PROGBITS        00000000 000498 000132 00       0   0  1
  [24] .shstrtab         STRTAB          00000000 0005ca 0000be 00       0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00394 0x00394 R E 0x1000
  LOAD           0x000394 0x08049394 0x08049394 0x00104 0x00108 RW  0x1000
  DYNAMIC        0x0003a4 0x080493a4 0x080493a4 0x000c8 0x000c8 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp

02      .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .in
it .plt .text .fini .rodata
   03     .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
   04     .dynamic
   05     .note.ABI-tag

Dynamic segment at offset 0x3a4 contains 20 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x804822c
 0x0000000d (FINI)                       0x8048364
 0x00000004 (HASH)                       0x8048128
 0x00000005 (STRTAB)                     0x80481a0
 0x00000006 (SYMTAB)                     0x8048150
 0x0000000a (STRSZ)                      74 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
```

```
 0x00000003 (PLTGOT)                     0x8049480
 0x00000002 (PLTRELSZ)                   16 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x804821c
 0x00000011 (REL)                        0x8048214
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x80481f4
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x80481ea
 0x00000000 (NULL)                       0x0

Relocation section '.rel.dyn' at offset 0x214 contains 1 entries:
 Offset     Info    Type                Sym.Value  Sym. Name
08049494  00000406 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x21c contains 2 entries:
 Offset     Info    Type                Sym.Value  Sym. Name
0804948c  00000107 R_386_JUMP_SLOT   08048254   puts
08049490  00000207 R_386_JUMP_SLOT   08048264   __libc_start_main

There are no unwind sections in this file.

Symbol table '.dynsym' contains 5 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048254   377 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
     2: 08048264   216 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@GLIBC_2.0 (2)
     3: 08048384     4 OBJECT  GLOBAL DEFAULT   14 _IO_stdin_used
     4: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__

Histogram for bucket list length (total of 3 buckets):
 Length  Number     % of total  Coverage
      0  0          (  0.0%)
      1  2          ( 66.7%)    50.0%
      2  1          ( 33.3%)   100.0%

Version symbols section '.gnu.version' contains 5 entries:
 Addr: 00000000080481ea  Offset: 0x0001ea  Link: 4 (.dynsym)
  000:   0 (*local*)       2 (GLIBC_2.0)       2 (GLIBC_2.0)       1 (*global*)
  004:   0 (*local*)

Version needs section '.gnu.version_r' contains 1 entries:
 Addr: 0x00000000080481f4  Offset: 0x0001f4  Link to section: 5 (.dynstr)
  000000: Version: 1  File: libc.so.6  Cnt: 1
```

```
  0x0010: Name: GLIBC_2.0   Flags: none   Version: 2
```

发现可执行文件中有好多的.section，都是有用的吗？我们试着除掉一些看起来没用的看看效果：

```
<onlyforos>[/home/onlyforos/hello]%objcopy -R .comment -R .note.ABI-tag –R .gnu.version -R .gnu.version_r hes hes1
<yug>[/home/yug/tmp/test]%ll hes*
-rwxr-xr-x      1 yug       yug             2672   6 月  22 17:01 hes*
-rwxr-xr-x      1 yug       yug             2156   6 月  22 17:22 hes1*
```

呵呵，小了不少。那么，一个 HelloWorld 的可执行在想象中应该是多大呢？让我们来估计一下。首先一个 ELF 头 52 字节。需要一个 text 段一个 data 段。一个段头为 32 字节，2 个为 64 字节。数据段内容也就是“ Hello,world ”大小为 12 字节，看过汇编代码，文本段内容估计 20-40 字节。这样估算下来，一个 Helloworld 的可执行文件看起来需要大约 150 左右字节就可以了。不知是不是这么一回事呢？怎样才能进一步的减小可执行文件的大小，让我们接着来进行尝试。

首先，我们可能会想到，调用库函数可能会有较大开销。库函数最终也会进行系统调用，那么，如果我们不调用库函数而直接调用系统调用，可执行文件大小会不会有所变化？
于是，不再调用 printf，而改为直接调用 write 系统调用。

```
<onlyforos>[/home/onlyforos/hello]%cat hw1.c
//#include <asm/unistd.h>

main()
{
        write(0, "hello,world\n", 12);
        return 0;
}
<onlyforos>[/home/onlyforos/hello]%gcc -o hw1 hw1.c
<onlyforos>[/home/onlyforos/hello]%hw1
hello,world
<onlyforos>[/home/onlyforos/hello]%ll hw1
-rwxr-xr-x      1 onlyforos onlyforos       11359   6 月   9 15:28 hw1*
```

呵呵，文件小了一些。让我们把符号信息去除再看看。

```
<onlyforos>[/home/onlyforos/hello]%gcc -s -o hw1 hw1.c
<onlyforos>[/home/onlyforos/hello]%ll hw1
-rwxr-xr-x      1 onlyforos onlyforos        2692   6 月   9 15:30 hw1*
```

还是很大。从《Hello how to World》中可以看到其实一个 Hello World 的真正入口并不是 Main 函数。main 函数的存在对于可执行文件的大小也会有一些开销的。所以，我们可以尝试没有 main 函数的 Hello World。Gcc 或 Ld 的-e 选项会使用指定的符号作为程序的初始执行点。-nostartfiles 会指明在链接时并不使用系统标准启动文件(Do not use the standard system startup files when linking.)。

```
<onlyforos>[/home/onlyforos/hello]%cat hw2.c
//#include <asm/unistd.h>
myfun()
{
```

```
    write(0, "Hello,world\n", 12);
    exit(0);
}
<onlyforos>[/home/onlyforos/hello]%gcc -s -nostartfiles -emyfun -o hw2 hw2.c
<onlyforos>[/home/onlyforos/hello]%hw2
Hello,world
<onlyforos>[/home/onlyforos/hello]%ll hw2
-rwxr-xr-x      1 onlyforos onlyforos       1540    6 月    9 15:32 hw2*
<onlyforos>[/home/onlyforos/hello]%gcc -s -O3 -nostartfiles -emyfun -o hw2 hw2.c
<onlyforos>[/home/onlyforos/hello]%ll hw2
-rwxr-xr-x      1 onlyforos onlyforos       1536    6 月    9 15:32 hw2*
<onlyforos>[/home/onlyforos/hello]%
```

从上可看到，优化起到了 4 字节的效果，说明到了一定程度对指令优化的作用越来越小了。经过这些处理，我们的可执行文件再创新低。不过，感觉还是不够。但对于只剩两行的 C 语言程序，真不知还能怎么处理。只是觉得此时 C 语言还是太高级了。还是转向汇编试试，看看能不能去除一些也许是 C 语言带来额外开销。我们可以使用 gcc –S 选项得到相应的汇编代码，当然也可以手工写一段汇编代码。

```
<onlyforos>[/home/onlyforos/hello]%cat h1.s
#h1.s
.section .data
output:
        .asciz "hello,world\n"
.section .text
.globl main
main:
push $output
call printf
push $0
call exit
```

程序中的.asciz 和.ascii 不同，在一个字符串末尾添加空字符，作用和.string 一样。编译执行。

```
<onlyforos>[/home/onlyforos/hello]%as -o h1.o h1.s
<onlyforos>[/home/onlyforos/hello]%ld -o h1 h1.o
h1.o: In function `_start':
h1.o(.text+0x6): undefined reference to `printf'
h1.o(.text+0xd): undefined reference to `exit'
<onlyforos>[/home/onlyforos/hello]%
```

链接时出错。这是因为在汇编语言程序中使用 C 库函数时，必须把 C 库文件连接到程序的目标代码中。我们使用的 printf 函数在动态库 libc.so 中，所以得使用-l 选项指出：-lc(只取 libX.so 中的 X)。但这还不够，问题在于连接器能够解析 C 函数，但函数本身并没有包含在可执行文件中，它是在程序执行时动态加载的。所以我们还必须指出加载动态库的程序。使用-dynamic-linker /lib/ld-linux.so.2 指定。

```
<onlyforos>[/home/onlyforos/hello]%as -o h1.o h1.s
<onlyforos>[/home/onlyforos/hello]%ld -dynamic-linker /lib/ld-linux.so.2 -o h1 -lc h1.o
```

```
<onlyforos>[/home/onlyforos/hello]%h1
hello,world
<onlyforos>[/home/onlyforos/hello]%ll h1
-rwxr-xr-x       1 onlyforos onlyforos        2680    6 月    9 16:08 h1*
```

没什么大的效果。去除 main 函数。使用程序默认的__start 入口。

```
<onlyforos>[/home/onlyforos/hello]%cat h2.s
#h.s
.section .data
output:
        .asciz "hello,world\n"
.section .text
.globl _start
_start:
push $output
call printf
push $0
call exit

<onlyforos>[/home/onlyforos/hello]%as -o h2.o h2.s
<onlyforos>[/home/onlyforos/hello]%ld -dynamic-linker /lib/ld-linux.so.2 -o h2 -lc h2.o
<onlyforos>[/home/onlyforos/hello]%h2
hello,world
<onlyforos>[/home/onlyforos/hello]%ll h2
-rwxr-xr-x       1 onlyforos onlyforos        1992    6 月    9 16:11 h2*
<onlyforos>[/home/onlyforos/hello]%ld -s -dynamic-linker /lib/ld-linux.so.2 -o h2 -lc h2.o
<onlyforos>[/home/onlyforos/hello]%ll h2
-rwxr-xr-x       1 onlyforos onlyforos        1376    6 月    9 16:12 h2*
<onlyforos>[/home/onlyforos/hello]%h2
hello,world
```

当然，直接使用 gcc 编译也是可以的。

```
<onlyforos>[/home/onlyforos/hello]%gcc -s -nostartfiles -o h2a h2.s
<onlyforos>[/home/onlyforos/hello]%ll h2a
-rwxr-xr-x       1 onlyforos onlyforos        1376    6 月    9 16:12 h2a*
<onlyforos>[/home/onlyforos/hello]%h2a
hello,world
```

1376 字节。是目前为止最小的尺寸了。不过通过前面的分析我们还有继续尝试的思路，那就是，不使用库函数，直接使用系统调用。

IA32 体系结构寄存器资源匮乏，EAX 寄存器用于保存要执行的系统调用值，而 EIP、EBP、ESP 都各有特定用途，所以对于系统调用就只能使用剩下的 5 个寄存器来保存输入值，这也是系统调用参数通常都不会超过 5 个的原因。需要超过 6 个参数的系统使用不同的方法把参数传递给系统调用。EBX 寄存器用于保存指向输入参数的内存位置的指针，输入参数按照连续的顺序存储。系统调用使用这个指针访问内存位置以便读取参数。通常系统调用时输入值顺序如下：

EBX(第一个参数)，ECX(第二个参数)，EDX(第三个参数)，ESI(第四个参数)，EDI(第五个

参数)。

所以，调用 write 系统调用时，首先把 write 的系统调用号 4 放入 EAX 寄存器，然后分别把参数放入相应的寄存器中。然后调用 int $0x80。调用号为 1 的是系统调用 exit。

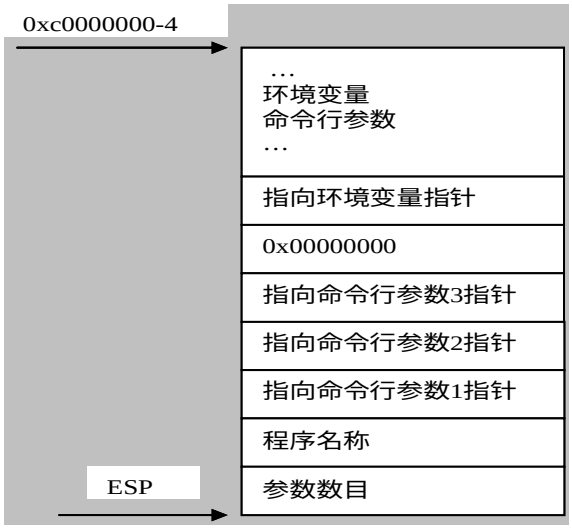
```
<onlyforos>[/home/onlyforos/hello]%cat h3.s
#hello.s
.section .data
output:
        .asciz "hello,world\n"
.section .text
.globl _start
_start:
movl $4, %eax
movl $1, %ebx
movl $output, %ecx
movl $13, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80

<onlyforos>[/home/onlyforos/hello]%as -o h3.o h3.s
<onlyforos>[/home/onlyforos/hello]%ld -o h3 h3.o
<onlyforos>[/home/onlyforos/hello]%h3
hello,world
<onlyforos>[/home/onlyforos/hello]%ll h3
-rwxr-xr-x       1 onlyforos onlyforos          723    6 月    9 16:17 h3*
<onlyforos>[/home/onlyforos/hello]%ld -s -o h3a h3.o
<onlyforos>[/home/onlyforos/hello]%h3a
hello,world
<onlyforos>[/home/onlyforos/hello]%ll h3a
-rwxr-xr-x       1 onlyforos onlyforos          396    6 月    9 16:17 h3a*
```

哈哈，去掉符号信息后文件只剩 396 字节!离我们的目标越来越近了。怎么样才能进一步裁剪呢。也许有的同学会说，我们只希望调用 printf，为什么要调用 exit 呢，把 exit 系统调用去掉应该会省下一些字节。

```
<onlyforos>[/home/onlyforos/hello]%as -o h4.o h4.s
<onlyforos>[/home/onlyforos/hello]%ld -o h4 h4.o
<onlyforos>[/home/onlyforos/hello]%h4
hello,world
Segmentation fault (core dumped)
```

却出现熟悉的段错误。这是因为我们把_start 当作它好像是一个 C 函数，并且试图从它返回。实际上，它根本不是一个函数。它只是目标文件中链接器用来定位程序入口点的一个符号。当我们的程序被激活时，它被直接激活。在《Hello how to world》中显示了在程序入口点也即__start 处栈布局为：



我们将会发现栈顶上是数 1，也就是是我们程序的 argc 值，这显然不像是一个地址，在栈顶上也没有返回地址。当程序把这个当作函数返回地址时就会报段错误了。段错误通常都是因为解除引用一个未初始化或非法值的指针引起的。那么__start 如何返回？正常情况下，__start 处将调用__libc_start_main，查看它的代码(glibc-2.3\sysdeps\generi\libc-start.c)我们可以看到，在这个函数中将会调用我们熟悉的 main 函数，main 函数返回后，接着调用 exit()退出。exit 的作用就是真正结束该进程，不再从这个函数返回。简单的说来，在 exit 调用中，该进程释放了所有的资源后通知父进程，然后调用 schedule()切换到其它进程，而由于自身状态已经是 TASK_ZOMBIE，一个“僵尸”，将再也不会得到调度。当父进程收到其信号将其剩余的资源全部释放掉。此时，该进程就真正的消失了。此话题与主题关系不大，不再赘述。总之，这几行代码是省略不得的。

到此时，还想拥有更小的可执行文件，就感觉汇编语言也有些“高级”了。那么让我们开始研究一下可执行文件吧。

```
<onlyforos>[/home/onlyforos/hello]%readelf -a h3a
ELF Header:
Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF32
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           Intel 80386
Version:                           0x1
Entry point address:               0x8048074
Start of program headers:          52 (bytes into file)
Start of section headers:          196 (bytes into file)
Flags:                             0x0
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
```

```
Number of program headers:          2
Size of section headers:            40 (bytes)
Number of section headers:          5
Section header string table index:  4


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        08048074 000074 000022 00  AX  0   0  4
  [ 2] .data             PROGBITS        08049098 000098 00000d 00  WA  0   0  4
  [ 3] .bss              NOBITS          080490a8 0000a8 000000 00  WA  0   0  4
  [ 4] .shstrtab         STRTAB          00000000 0000a8 00001c 00      0   0  1
  Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)


Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x00096 0x00096 R E 0x1000
  LOAD           0x000098 0x08049098 0x08049098 0x0000d 0x00010 RW  0x1000

Section to Segment mapping:
 Segment Sections...
  00     .text
  01     .data

There is no dynamic segment in this file.
There are no relocations in this file.
There are no unwind sections in this file.
No version information found in this file.
```

从上面可以看到，除了 52 字节的 ELF 头，现在可执行文件中有两个可载入的程序段，.text 文本段及.data 数据段。一个程序运行，这两个载入的段一般来说是必不可少的。此外还有 4 个节头表。尽管.bss 节表大小为 0，但还是占可执行文件字节的，.shstrtab 节表保存着大小为 0x1c 的 section 名称。可以提出设想，既然那两个程序段必不可少，但可不可以把这几个节头表除掉呢，因为感觉它们和程序运行没有太大关系。不过，想修改二进制文件中的节头表，好象已经没有标准工具或命令来实现这个任务了。只能自己想办法了。我们采用 Ultra-edit 编辑器的二进制文件编辑功能来修改删除(ctrl+x)这个文件的内容。

首先，我们打开这个可执行文件，来查看其内容。

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ;  ELF............
00000010h: 02 00 03 00 01 00 00 00 74 80 04 08 34 00 00 00 ; ........t€..4...
00000020h: C4 00 00 00 00 00 00 00 34 00 20 00 02 00 28 00 ; ?.......4. ...(.
00000030h: 05 00 04 00 01 00 00 00 00 00 00 00 00 80 04 08 ; .............€..
00000040h: 00 80 04 08 96 00 00 00 96 00 00 00 05 00 00 00 ; .€..?..?......
00000050h: 00 10 00 00 01 00 00 00 98 00 00 00 98 90 04 08 ; ........?..槓..
00000060h: 98 90 04 08 0D 00 00 00 10 00 00 00 06 00 00 00 ; 槓.............
00000070h: 00 10 00 00 B8 04 00 00 00 BB 01 00 00 00 B9 98 ; ....?...?...徨
00000080h: 90 04 08 BA 0D 00 00 00 CD 80 B8 01 00 00 00 BB ; ?.?...蜮?...?
00000090h: 00 00 00 00 CD 80 00 00 68 65 6C 6C 6F 2C 77 6F ; ....蜮..hello,wo
000000a0h: 72 6C 64 0A 00 00 00 00 00 2E 73 68 73 74 72 74 ; rld.......shstrt
000000b0h: 61 62 00 2E 74 65 78 74 00 2E 64 61 74 61 00 2E ; ab..text..data..
000000c0h: 62 73 73 00 00 00 00 00 00 00 00 00 00 00 00 00 ; bss.............
000000d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000e0h: 00 00 00 00 00 00 00 00 00 00 00 00 0B 00 00 00 ; ................
000000f0h: 01 00 00 00 06 00 00 00 74 80 04 08 74 00 00 00 ; ........t€..t...
00000100h: 22 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 ; "...............
00000110h: 00 00 00 00 11 00 00 00 01 00 00 00 03 00 00 00 ; ................
00000120h: 98 90 04 08 98 00 00 00 0D 00 00 00 00 00 00 00 ; 槓..?..........
00000130h: 00 00 00 00 04 00 00 00 00 00 00 00 17 00 00 00 ; ................
00000140h: 08 00 00 00 03 00 00 00 A8 90 04 08 A8 00 00 00 ; ........ ..?..
00000150h: 00 00 00 00 00 00 00 00 00 00 00 00 04 00 00 00 ; ................
00000160h: 00 00 00 00 01 00 00 00 03 00 00 00 00 00 00 00 ; ................
00000170h: 00 00 00 00 A8 00 00 00 1C 00 00 00 00 00 00 00 ; ....?...........
00000180h: 00 00 00 00 01 00 00 00 00 00 00 00             ; ............
```

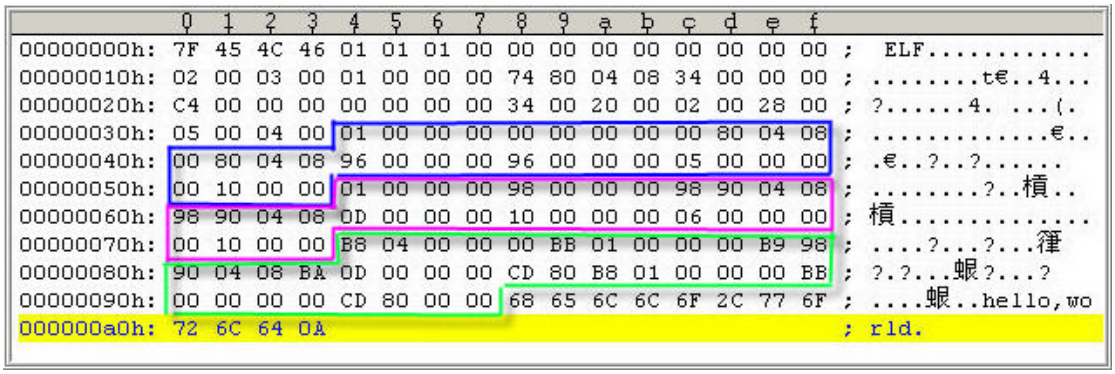
然后选中后面我们认为没用的节头表，删除。

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ;  ELF............
00000010h: 02 00 03 00 01 00 00 00 74 80 04 08 34 00 00 00 ; ........t€..4...
00000020h: C4 00 00 00 00 00 00 00 34 00 20 00 02 00 28 00 ; ?.......4. ...(.
00000030h: 05 00 04 00 01 00 00 00 00 00 00 00 00 80 04 08 ; .............€..
00000040h: 00 80 04 08 96 00 00 00 96 00 00 00 05 00 00 00 ; .€..?..?......
00000050h: 00 10 00 00 01 00 00 00 98 00 00 00 98 90 04 08 ; ........?..槓..
00000060h: 98 90 04 08 0D 00 00 00 10 00 00 00 06 00 00 00 ; 槓.............
00000070h: 00 10 00 00 B8 04 00 00 00 BB 01 00 00 00 B9 98 ; ....?...?...徨
00000080h: 90 04 08 BA 0D 00 00 00 CD 80 B8 01 00 00 00 BB ; ?.?...蜮?...?
00000090h: 00 00 00 00 CD 80 00 00 68 65 6C 6C 6F 2C 77 6F ; ....蜮..hello,wo
000000a0h: 72 6C 64 0A                                     ; rld.
```

不知修改后的文件还可不可以运行，保存后运行试试。

```
<onlyforos>[/home/onlyforos/hello]%he1
hello,world
<onlyforos>[/home/onlyforos/hello]%ll he1
-rwxr-xr-x    1 onlyforos onlyforos         164    6 月    9 16:17 he1*
```

哈哈，可以运行!看看大小，164 字节。几乎达到了我们预告的猜想了。那么，好奇的人们可以休息了吗？文件还可以再小一些吗？想进一步探询，还是让我们首先来进一步分析一下现在的可执行文件吧。



```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ;  ELF.............
00000010h: 02 00 03 00 01 00 00 00 74 80 04 08 34 00 00 00 ; ........t€..4...
00000020h: C4 00 00 00 00 00 00 00 34 00 20 00 02 00 28 00 ; ?.......4. ...(.
00000030h: 05 00 04 00 01 00 00 00 00 00 00 00 00 80 04 08 ; .............€..
00000040h: 00 80 04 08 96 00 00 00 96 00 00 00 05 00 00 00 ; .€..?..?.......
00000050h: 00 10 00 00 01 00 00 00 98 00 00 00 98 90 04 08 ; ........?..槓..
00000060h: 98 90 04 08 0D 00 00 00 10 00 00 00 06 00 00 00 ; 槓..............
00000070h: 00 10 00 00 B8 04 00 00 00 BB 01 00 00 00 B9 98 ; ....?...?...徨
00000080h: 90 04 08 BA 0D 00 00 00 CD 80 B8 01 00 00 00 BB ; ?.?...蜾?...?
00000090h: 00 00 00 00 CD 80 00 00 68 65 6C 6C 6F 2C 77 6F ; ....蜾..hello,wo
000000a0h: 72 6C 64 0A                                     ; rld.
```

如图中所示，首先的 52 字节就是可执行文件必须有的 ELF 头，从头中我们可以得知程序段头表的大小为 32 字节，有 2 个程序段表。节表的大小为 40 字节，有 5 个节表。但实际上目前的文件中的节表已经被残酷的删除了，所以 ELF 头中的该字段数据已经没有意义。ELF 头后跟着 2 个程序段表，蓝色框中为一可加载程序段，表示将文件的 0x96 字节数据加载到虚拟地址 0x8048000 处。即主要是我们文本段代码指令。当然从 ELF 头中我们也得知，程序是从 0x8048074 处开始执行的。粉色框就是可加载的数据段，将文件中偏移为 98 处的 0x0D 字节数据加载到 0x8049098 处。绿框中就是我们的代码指令，之后，就是数据段内容。

如此分析下来，感觉剩下的数据几乎没有办法再裁剪了。但俗话说：人有多大胆，地有多大产(这是俗话吗)。心有不甘地再看上一遍这些 0 和 1。突然发现，我们的代码指令中怎么那么多的 0 啊。

```
b8 04 00 00 00 bb 01 00 00 00 b9 98 90 04 08 ba 0d 00 00 00 cd 80 b8 01 00 00 00 bb 00 00 00 00 cd 80
```

34 字节数据中居然有 15 字节的 0。是不是可以进行一些修改呢？我们来看一下其代码：

```
>[/home/onlyforos/hello]%objdump -d h3a

h3a:     file format elf32-i386

objdump: h3a: no symbols
Disassembly of section .text:

08048074 <.text>:
 8048074:       b8 04 00 00 00          mov    $0x4,%eax
 8048079:       bb 01 00 00 00          mov    $0x1,%ebx
 804807e:       b9 98 90 04 08          mov    $0x8049098,%ecx
 8048083:       ba 0d 00 00 00          mov    $0xd,%edx
 8048088:       cd 80                   int    $0x80
 804808a:       b8 01 00 00 00          mov    $0x1,%eax
 804808f:       bb 00 00 00 00          mov    $0x0,%ebx
 8048094:       cd 80                   int    $0x80
```

可以看到，比如把系统调用号 4 放入寄存器 EAX 中的指令为 b8 04 00 00 00，但实际上我们可能只需要把 4 放入 AL(32 位寄存器 EAX 的低 8 位寄存器)中就行了。这样，只需要一个字节而不是 4 个字节的立即数，从而就可以精减我们的指令字节数。按照这个思路，重新写汇编代码，编译后用 OBJDUMP 查看指令。当然也可以直接查看 IA32 指令手册直接写机器指令。

```
<onlyforos>[/home/onlyforos/hello]%cat he2.s
#hello.s
.section .data
output:
        .asciz "hello,world\n"
.section .text
.globl _start
_start:
movb $4, %al
movb $1, %bl
movl $output, %ecx
movb $12, %dl
int $0x80
movb $1, %al
movb $0, %bl
int $0x80

<onlyforos>[/home/onlyforos/hello]%as -o he2.o he2.s
<onlyforos>[/home/onlyforos/hello]%ld -o he2 he2.o
<onlyforos>[/home/onlyforos/hello]%he2
hello,world
<onlyforos>[/home/onlyforos/hello]%objdump –d he2 >> he2.txt
<onlyforos>[/home/onlyforos/hello]%cat he2.txt

he2:        file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:    b0 04                   mov    $0x4,%al
 8048076:    b3 01                   mov    $0x1,%bl
 8048078:    b9 88 90 04 08          mov    $0x8049088,%ecx
 804807d:    b2 0c                   mov    $0xc,%dl
 804807f:    cd 80                   int    $0x80
 8048081:    b0 01                   mov    $0x1,%al
 8048083:    b3 00                   mov    $0x0,%bl
 8048085:    cd 80                   int    $0x80
```

可以看到指令代码从 34 字节变为 19 字节。又少了 15 字节。把它更新到我们可执行文件试试看。

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 ;  ELF............
00000010h: 02 00 03 00 01 00 00 00 74 80 04 08 34 00 00 00 ; ........t€..4...
00000020h: C4 00 00 00 00 00 00 00 34 00 20 00 02 00 28 00 ; ?.......4. ...(.
00000030h: 05 00 04 00 01 00 00 00 00 00 00 00 00 80 04 08 ; .............€..
00000040h: 00 80 04 08 96 00 00 00 96 00 00 00 05 00 00 00 ; .€..?..?......
00000050h: 00 10 00 00 01 00 00 00 87 00 00 00 87 90 04 08 ; ........?..聒..
00000060h: 87 90 04 08 0D 00 00 00 10 00 00 00 06 00 00 00 ; 聒.............
00000070h: 00 10 00 00 B0 04 B3 01 B9 87 90 04 08 B2 0C CD ; ....??孰?.??
00000080h: 80 B0 01 B3 00 CD 80 68 65 6C 6C 6F 2C 77 6F 72 ; €??蜾hello,wor
00000090h: 6C 64 0A                                       ; ld.
```

由于指令字节少了 15 字节，数据段数据在文件中的位置发生了变化，偏移值从以前的 0x98 处变为现在的 0x87 处，所以除了修改指令，我们还需要修改程序段的数据偏移及指令中的数据地址值。修改后我们运行程序：

```
<onlyforos>[/home/onlyforos/hello]%he3
hello,world
<onlyforos>[/home/onlyforos/hello]%ll he3
-rwxr-xr-x       1 onlyforos onlyforos          147    6 月    9 16:17 he3*
```

至此，几乎可以肯定地说，这么小的 HelloWorld 已经完全达到了我们预先的猜想。但可不可以再小一些呢？

想到对于一个 147 字节的可执行文件，一个 52 字节的 ELF 头绝对算是大的开销了。那么，这些字节都用到了吗？此处省略搬出 ELF 头结构进行分析的若干行。最后的结论，我们可以利用 ELF 头的一些字段，把数据段干掉。也不管有没有或是什么道理了，准则只剩一个，那就是可以执行看到：Hello,world。于是：

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 7F 45 4C 46 68 65 6C 6C 6F 2C 77 6F 72 6C 64 0A ;  ELFhello,world.
00000010h: 02 00 03 00 01 00 00 00 54 80 04 08 34 00 00 00 ; ........T€..4...
00000020h: AA AA AA AA AA AA AA AA AA AA 20 00 01 00 AA AA ;              ...
00000030h: AA AA AA AA 01 00 00 00 00 00 00 00 00 80 04 08 ;     .........€..
00000040h: 00 80 04 08 76 00 00 00 76 00 00 00 05 00 00 00 ; .€..v...v.......
00000050h: 00 10 00 00 B0 04 B3 01 B9 04 80 04 08 B2 0C CD ; ....???€..??
00000060h: 80 B0 01 B3 00 CD 80                           ; €??蜾
```

```
<onlyforos>[/home/onlyforos/hello]%he4
hello,world
<onlyforos>[/home/onlyforos/hello]%ll he4
-rwxr-xr-x       1 onlyforos onlyforos          103    6 月    9 16:17 he4*
```

可以看到，把数据段嵌到 ELF 头，修改 ELF 头中的程序入口地址，修改了指令中数据地址值。并以 A 修改了 ELF 貌似没用的字段。结果仍可以正常运行并且大小居然缩小到 103 字节。

那么，还可以小吗？

如图中看到，既然值的 A 的地方可能是没用的地方，那么，我们就可以继续利用这些空间：

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 7F 45 4C 46 68 65 6C 6C 6F 2C 77 6F 72 6C 64 0A ;  ELFhello,world.
00000010h: 02 00 03 00 AA AA AA AA 4E 80 04 08 2E 00 00 00 ; ....    N€......
00000020h: AA AA AA AA AA AA AA AA AA AA 20 00 01 00 01 00 ; ..........  .....
00000030h: 00 00 00 00 00 00 00 80 04 08 00 80 04 08 76 00 ; .......€...€..v.
00000040h: 00 00 76 00 00 00 05 00 00 00 00 10 00 00 B0 04 ; ..v...........?
00000050h: B3 01 B9 04 80 04 08 B2 0C CD 80 B0 01 B3 00 CD ; ??€..?蝯???
00000060h: 80                                              ; €
```

```
<onlyforos>[/home/onlyforos/hello]%he5
hello,world
<onlyforos>[/home/onlyforos/hello]%ll he5
-rwxr-xr-x       1 onlyforos onlyforos          97    6 月    9 16:17 he5*
```

如图把程序段往前移到了 ELF 头中，修改了头中入口地址。运行执行，Hello,world。哈哈，大小变为了 97 字节。

几乎不加思索的发问了：那么，还可以小吗？

既然 ELF 的字段可以利用，那么，程序段表中的字段应该也可以用吧(略去分析程序段表的若干行，其实分析的结果也是没什么结果，因为看起来应该是这样的东东其实也没这样)：

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 7F 45 4C 46 68 65 6C 6C 6F 2C 77 6F 72 6C 64 0A ;  ELFhello,world.
00000010h: 02 00 03 00 AA AA AA AA 3C 80 04 08 2E 00 00 00 ; ....    <€......
00000020h: AA AA AA AA AA AA AA AA AA AA 20 00 01 00 01 00 ; ..........  .....
00000030h: 00 00 00 00 00 00 00 80 04 08 00 80 B0 04 B3 01 ; .......€...€??
00000040h: B9 04 80 04 08 B2 0C CD 80 B0 01 B3 00 CD 80    ; ?€..?蝯??蝯
```

```
<onlyforos>[/home/onlyforos/hello]%he6
hello,world
<onlyforos>[/home/onlyforos/hello]%ll he6
-rwxr-xr-x       1 onlyforos onlyforos          79    6 月    9 16:17 he6*
```

总之，79 字节!还有 14 字节的 A 没利用，此外，linux 默认加载地址为 0x8048000，此地址可以在 LD 代码中脚本中找到(TEXT_START_ADDR=0x08048000)。其实，加载地址并不一定非得从这个地址开始，所以感觉程序入口地址及加载虚地址还可以同时改变。

热情不足了。此时，好象只需要不停地尝试就可以了。

那么，还可以小吗？还可以小吗？可以小吗？小吗？吗？☺……