（未完成）catalyst-system WriteUp（2019暑假CTF第一周reverse）

转载

afu42832 于 2019-07-22 16:08:00 发布 122 收藏
文章标签： 操作系统 数据结构与算法
原文链接：http://www.cnblogs.com/hardcoreYutian/p/11193928.html
版权
目录

预备学习——Linux实践：ELF文件格式分析

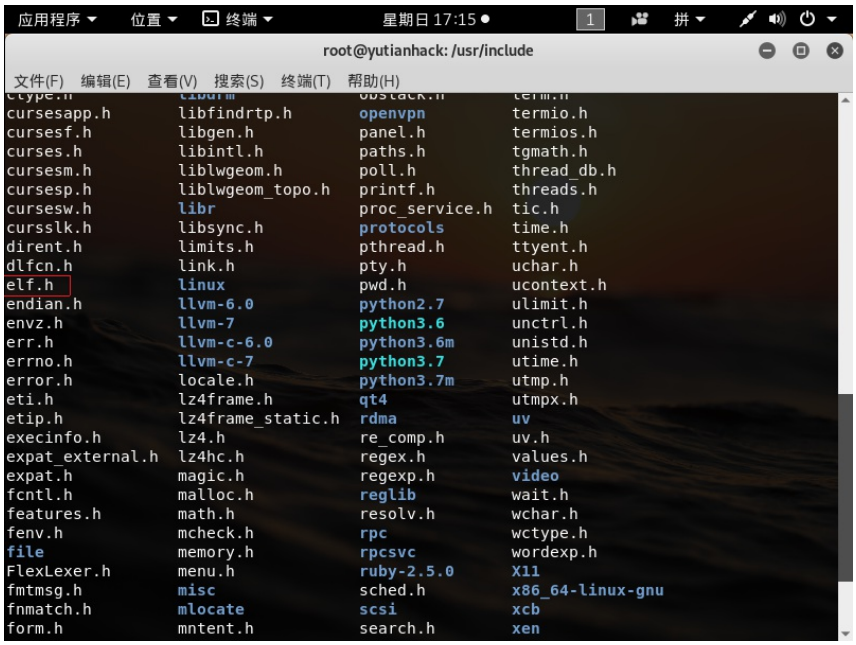
一、概述

1.ELF全称Executable and Linkable Format，可执行连接格式，ELF格式的文件用于存储Linux程序。ELF文件（目标文件）格式主要有三种：

- 可重定向文件（Relocatable file）：文件保存着代码和适当的数据，用来和其他的目标文件一起来创建一个可执行文件或者是一个共享目标文件（目标文件或者静态库文件，即Linux通常后缀为.a和.o的文件）。
- 可执行文件（Executable file）：文件保存着一个用来执行的程序（例如bash，gcc等）。
- 共享目标文件：动态库文件（Linux下后缀为.so的文件）。文件保存着代码和合适的数据（这些数据是在连接时候被连接器ld和运行时动态连接器使用的）。

二、分析ELF文件头（ELF header）

- 在终端输入`cd /usr/include`进入include目录后查看elf.h文件，查看ELF的文件头ELF Header的定义。



（这里以64位的为例，64位和32位的都可以查看到）

```
typedef struct
{
  unsigned char   e_ident[EI_NIDENT];   /* Magic number and other info */
  Elf64_Half   e_type;            /* Object file type */
  Elf64_Half   e_machine;     /* Architecture */
  Elf64_Word   e_version;     /* Object file version */
  Elf64_Addr   e_entry;       /* Entry point virtual address */
  Elf64_Off    e_phoff;       /* Program header table file offset */
  Elf64_Off    e_shoff;       /* Section header table file offset */
  Elf64_Word   e_flags;       /* Processor-specific flags */
  Elf64_Half   e_ehsize;      /* ELF header size in bytes */
  Elf64_Half   e_phentsize;       /* Program header table entry size */
  Elf64_Half   e_phnum;       /* Program header table entry count */
  Elf64_Half   e_shentsize;       /* Section header table entry size */
  Elf64_Half   e_shnum;       /* Section header table entry count */
  Elf64_Half   e_shstrndx;        /* Section header string table index */
} Elf64_Ehdr;
```

关于其长度信息：

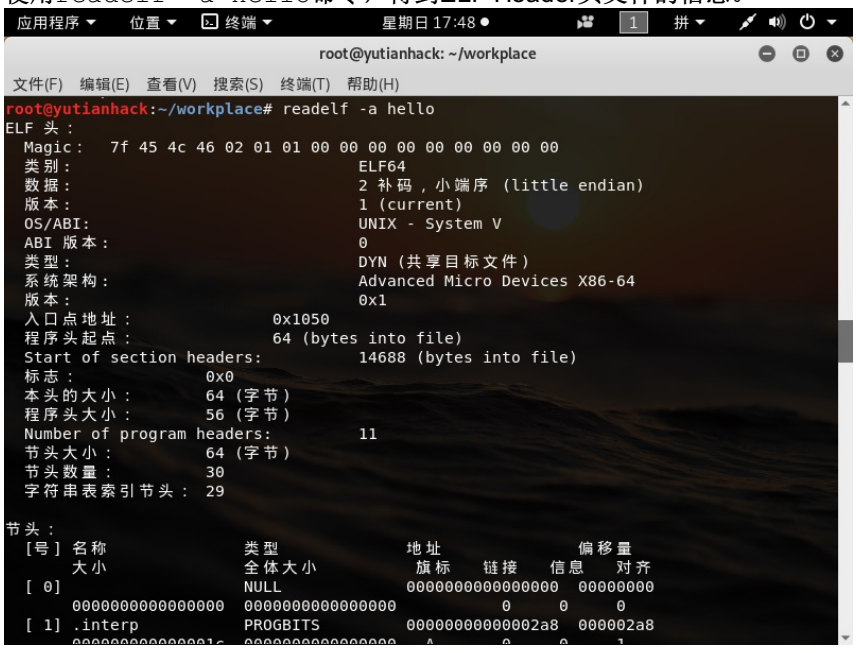
插入图片elf5.pdf

可以看到32位与64位的部分长度不同，要根据具体文件具体判断。

- 写一个小程序并编译，生成hello可执行文件

```
root@yutianhack:~/workplace# gedit hello.c
root@yutianhack:~/workplace# gcc hello.c -o hello
root@yutianhack:~/workplace# ./hello
hello 20190714root@yutianhack:~/workplace#
```

使用`readelf -a hello`命令，得到ELF Header头文件的信息。



- 通过上图信息，可以得出Elf Header的Size为64bytes，所以可以使用hexdump工具将头文件的16进制表打开。
使用`hexdump -x hello -n 52`命令查看hello文件头的16进制表（前52bytes），对格式进行分析。

```
root@yutianhack:~/workplace# hexdump -x hello -n 64
0000000    457f    464c    0102    0001    0000    0000    0000    0000
0000010    0003    003e    0001    0000    1050    0000    0000    0000
0000020    0040    0000    0000    0000    3960    0000    0000    0000
0000030    0000    0000    0040    0038    000b    0040    001e    001d
0000040
```

- 第一行
  - 对应e_ident[EI_NIDENT]。实际表示内为7f454c4602010001000000000000000，前四个字节7f454c46（0x45，0x4c，0x46是'e'，'l'，'f'对应的ascii编码）是一个魔数，表示这是一个ELF对象。
  - 接下来的一个字节02表示是一个64位对象，接下来的一个字节01表示是小端法表示，再接下来的一个字节01表示文件头版本。剩下的都默认为0。

- 第二行
  - e_type（2字节）值为0003（不确定写得对不对），表示是一个共享目标文件。
  - e_machine（2字节）值为003e（不确定写得对不对），表示是Advanced Micro Devices X86-64架构。
  - e_version（4字节）值为00010000（不确定写得对不对），表示是当前版本。
  - e_entry（8字节）值为1050000000000000（不确定写得对不对），表示入口点地址。

- 第三行
  - e_phoff（8字节）值为0040000000000000（不确定写得对不对），表示程序头表的偏移地址。
  - e_shoff（8字节）值为3960000000000000不确定写得对不对），表示段表的偏移地址。

- 第四行
  - e_flags（4字节）值为00000000（不确定写得对不对），表示未知处理器特定标志#define EF_SH_UNKNOWN 0x0。
  - e_ehsize（2字节）值为0040（不确定写得对不对），表示.elf文件头大小为00 40（64个字节）。
  - e_phentsize（2字节）值为0038（不确定写得对不对），表示一个程序头表中的入口的长度。
  - e_phnum（2字节）值为000b（不确定写得对不对），表示程序头表中的入口数目。
  - e_shentsize（2字节）值为0040（64字节）（不确定写得对不对），表示section header table中每个header的大小。
  - e_shnum（2字节）值为001e（不确定写得对不对），表示段表入口有30个（不确定），即段表有13段。
  - e_shstrndx（2字节）值为001d（不确定写得对不对），表示段表字符串在段表中的索引号。为29（不确定）。
  本部分最大的问题是怎么划分哪几位代表什么。经过一番波折找到了64位和32位占字节不一的问题，但是小端法是否反序读，高位0是否省略等问题还是不清楚。参考了多位学长学姐的博客，格式各有不同，最终没有搞明白这一点。我采用的方法是全都没有反序，按照终端显示的顺序来读。

三、通过文件头找到section header table，理解其内容

- `file elf`文件的名字显示生成的目标文件hello的类型

```
root@yutianhack:~/workplace# file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically link
ed, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=
00a518a80cd9cec60893b6a94438e3986b152941, not stripped
```

- hello是一个可执行文件，输入`ls -l hello`查看hello的大小

```
root@yutianhack:~/workplace# ls -l hello
-rwxr-xr-x 1 root root 16608 7月   14 17:37 hello
```

- 如上图，hello大小为16608字节。输入`hexdump -x hello`用16进制的数字显示hello的内容。（其中，第二列是16进制表示的偏移地址）

```
-rwxr-xr-x 1 root root 16608 7月  14 17:37 hello
root@yutianhack:~/workplace# hexdump -x hello
0000000    457f    464c    0102    0001    0000    0000    0000    0000
0000010    0003    003e    0001    0000    1050    0000    0000    0000
0000020    0040    0000    0000    0000    3960    0000    0000    0000
0000030    0000    0000    0040    0038    000b    0040    001e    001d
0000040    0006    0000    0004    0000    0040    0000    0000    0000
0000050    0040    0000    0000    0000    0040    0000    0000    0000
0000060    0268    0000    0000    0000    0268    0000    0000    0000
0000070    0008    0000    0000    0000    0003    0000    0004    0000
0000080    02a8    0000    0000    0000    02a8    0000    0000    0000
0000090    02a8    0000    0000    0000    001c    0000    0000    0000
00000a0    001c    0000    0000    0000    0001    0000    0000    0000
00000b0    0001    0000    0004    0000    0000    0000    0000    0000
00000c0    0000    0000    0000    0000    0000    0000    0000    0000
00000d0    0568    0000    0000    0000    0568    0000    0000    0000
00000e0    1000    0000    0000    0000    0001    0000    0005    0000
00000f0    1000    0000    0000    0000    1000    0000    0000    0000
0000100    1000    0000    0000    0000    01cd    0000    0000    0000
0000110    01cd    0000    0000    0000    1000    0000    0000    0000
0000120    0001    0000    0004    0000    2000    0000    0000    0000
0000130    2000    0000    0000    0000    2000    0000    0000    0000
0000140    0158    0000    0000    0000    0158    0000    0000    0000
0000150    1000    0000    0000    0000    0001    0000    0006    0000
```

- 输入`objdump -x hello`来显示hello中各个段以及符号表的相关信息。

```
root@yutianhack:~/workplace# objdump -x hello

hello：     文件格式 elf64-x86-64
hello
体系结构：i386:x86-64，标志 0x00000150：
HAS_SYMS, DYNAMIC, D_PAGED
起始地址 0x0000000000001050

程序头：
    PHDR off    0x0000000000000040 vaddr 0x0000000000000040 paddr 0x000000000000
0040 align 2**3
         filesz 0x0000000000000268 memsz 0x0000000000000268 flags r--
  INTERP off    0x00000000000002a8 vaddr 0x00000000000002a8 paddr 0x000000000000
02a8 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x000000000000
0000 align 2**12
         filesz 0x0000000000000568 memsz 0x0000000000000568 flags r--
    LOAD off    0x0000000000001000 vaddr 0x0000000000001000 paddr 0x000000000000
1000 align 2**12
         filesz 0x00000000000001cd memsz 0x00000000000001cd flags r-x
    LOAD off    0x0000000000002000 vaddr 0x0000000000002000 paddr 0x000000000000
```

- 输入`readelf -a hello`来查看各个段信息。
  - ELF文件头信息：

```
root@yutianhack:~/workplace# readelf -a hello
ELF 头：
  Magic：   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别：                             ELF64
  数据：                             2 补码，小端序 (little endian)
  版本：                             1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本：                         0
  类型：                             DYN (共享目标文件)
  系统架构：                         Advanced Micro Devices X86-64
  版本：                             0x1
  入口点地址：               0x1050
  程序头起点：               64 (bytes into file)
  Start of section headers:          14688 (bytes into file)
  标志：             0x0
  本头的大小：       64 (字节)
  程序头大小：       56 (字节)
  Number of program headers:         11
  节头大小：         64 (字节)
  节头数量：         30
  字符串表索引节头： 29
```

  - 段表Section header table：

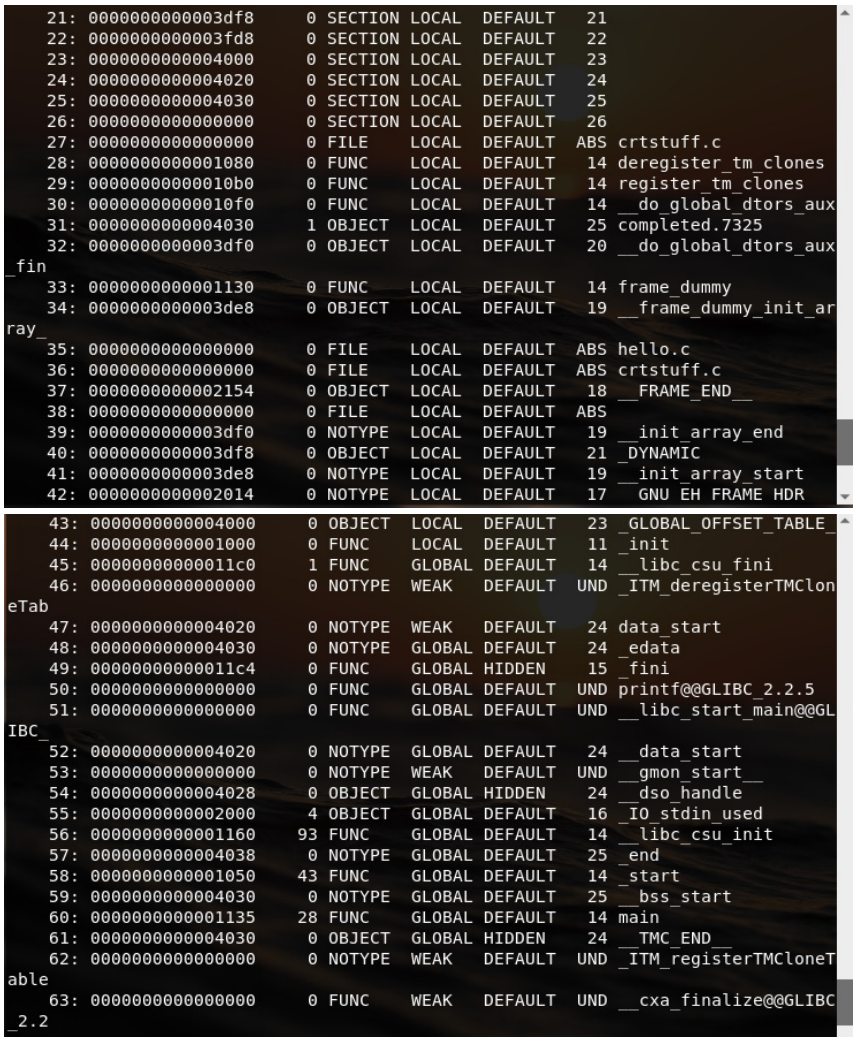
```
节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小          旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         00000000000002a8  000002a8
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             00000000000002c4  000002c4
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             00000000000002e4  000002e4
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000000308  00000308
       0000000000000024  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000000330  00000330
       00000000000000a8  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           00000000000003d8  000003d8
       0000000000000084  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           000000000000045c  0000045c
       000000000000000e  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000000470  00000470
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000000490  00000490
       00000000000000c0  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000000550  00000550
```

```
  [11] .init             PROGBITS         0000000000001000  00001000
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000001020  00001020
       0000000000000020  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000001040  00001040
       0000000000000008  0000000000000008  AX       0     0     8
  [14] .text             PROGBITS         0000000000001050  00001050
       0000000000000171  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         00000000000011c4  000011c4
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000002000  00002000
       0000000000000013  0000000000000000   A       0     0     4
  [17] .eh_frame_hdr     PROGBITS         0000000000002014  00002014
       000000000000003c  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         0000000000002050  00002050
       0000000000000108  0000000000000000   A       0     0     8
  [19] .init_array       INIT_ARRAY       0000000000003de8  00002de8
       0000000000000008  0000000000000008  WA       0     0     8
  [20] .fini_array       FINI_ARRAY       0000000000003df0  00002df0
       0000000000000008  0000000000000008  WA       0     0     8
  [21] .dynamic          DYNAMIC          0000000000003df8  00002df8
       00000000000001e0  0000000000000010  WA       6     0     8
  [22] .got              PROGBITS         0000000000003fd8  00002fd8
       0000000000000028  0000000000000008  WA       0     0     8
```

```
  [23] .got.plt          PROGBITS         0000000000004000  00003000
       0000000000000020  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         0000000000004020  00003020
       0000000000000010  0000000000000000  WA       0     0     8
  [25] .bss              NOBITS           0000000000004030  00003030
       0000000000000008  0000000000000000  WA       0     0     1
  [26] .comment          PROGBITS         0000000000000000  00003030
       000000000000001c  0000000000000001  MS       0     0     1
  [27] .symtab           SYMTAB           0000000000000000  00003050
       0000000000000600  0000000000000018          28    45     8
  [28] .strtab           STRTAB           0000000000000000  00003650
       0000000000000205  0000000000000000           0     0     1
  [29] .shstrtab         STRTAB           0000000000000000  00003855
       0000000000000107  0000000000000000           0     0     1
```

- 符号表Symbol table：

```
Symbol table '.symtab' contains 64 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000000002a8     0 SECTION LOCAL  DEFAULT    1
     2: 00000000000002c4     0 SECTION LOCAL  DEFAULT    2
     3: 00000000000002e4     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000308     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000330     0 SECTION LOCAL  DEFAULT    5
     6: 00000000000003d8     0 SECTION LOCAL  DEFAULT    6
     7: 000000000000045c     0 SECTION LOCAL  DEFAULT    7
     8: 0000000000000470     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000000490     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000000550     0 SECTION LOCAL  DEFAULT   10
    11: 0000000000001000     0 SECTION LOCAL  DEFAULT   11
    12: 0000000000001020     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000001040     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000001050     0 SECTION LOCAL  DEFAULT   14
    15: 00000000000011c4     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000002000     0 SECTION LOCAL  DEFAULT   16
    17: 0000000000002014     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000002050     0 SECTION LOCAL  DEFAULT   18
    19: 0000000000003de8     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000003df0     0 SECTION LOCAL  DEFAULT   20
```



```
    21: 0000000000003df8     0 SECTION LOCAL  DEFAULT   21
    22: 0000000000003fd8     0 SECTION LOCAL  DEFAULT   22
    23: 0000000000004000     0 SECTION LOCAL  DEFAULT   23
    24: 0000000000004020     0 SECTION LOCAL  DEFAULT   24
    25: 0000000000004030     0 SECTION LOCAL  DEFAULT   25
    26: 0000000000000000     0 SECTION LOCAL  DEFAULT   26
    27: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    28: 0000000000001080     0 FUNC    LOCAL  DEFAULT   14 deregister_tm_clones
    29: 00000000000010b0     0 FUNC    LOCAL  DEFAULT   14 register_tm_clones
    30: 00000000000010f0     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
    31: 0000000000004030     1 OBJECT  LOCAL  DEFAULT   25 completed.7325
    32: 0000000000003df0     0 OBJECT  LOCAL  DEFAULT   20 __do_global_dtors_aux
_fin
    33: 0000000000001130     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
    34: 0000000000003de8     0 OBJECT  LOCAL  DEFAULT   19 __frame_dummy_init_ar
ray_
    35: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.c
    36: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    37: 0000000000002154     0 OBJECT  LOCAL  DEFAULT   18 __FRAME_END__
    38: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    39: 0000000000003df0     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_end
    40: 0000000000003df8     0 OBJECT  LOCAL  DEFAULT   21 _DYNAMIC
    41: 0000000000003de8     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_start
    42: 0000000000002014     0 NOTYPE  LOCAL  DEFAULT   17   GNU EH FRAME HDR
    43: 0000000000004000     0 OBJECT  LOCAL  DEFAULT   23 _GLOBAL_OFFSET_TABLE_
    44: 0000000000001000     0 FUNC    LOCAL  DEFAULT   11 _init
    45: 00000000000011c0     1 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    46: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMClon
eTab
    47: 0000000000004020     0 NOTYPE  WEAK   DEFAULT   24 data_start
    48: 0000000000004030     0 NOTYPE  GLOBAL DEFAULT   24 _edata
    49: 00000000000011c4     0 FUNC    GLOBAL HIDDEN    15 _fini
    50: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    51: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GL
IBC_
    52: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
    53: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    54: 0000000000004028     0 OBJECT  GLOBAL HIDDEN    24 __dso_handle
    55: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    56: 0000000000001160    93 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    57: 0000000000004038     0 NOTYPE  GLOBAL DEFAULT   25 _end
    58: 0000000000001050    43 FUNC    GLOBAL DEFAULT   14 _start
    59: 0000000000004030     0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
    60: 0000000000001135    28 FUNC    GLOBAL DEFAULT   14 main
    61: 0000000000004030     0 OBJECT  GLOBAL HIDDEN    24 __TMC_END__
    62: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneT
able
    63: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@@GLIBC
_2.2
```

四、通过section header table找到各section

在一个ELF文件中有一个section header table，通过它我们可以定位到所有的section，而ELF header中的e_shoff变量就是保存section header table入口对文件头的偏移量。而每个section都会对应一个section header，所以只要在section header table中找到每个section header，就可以通过section header找到你想要的section。

下面以可执行文件hello为例，以保存代码段的section为例来讲解读取某个section的过程。

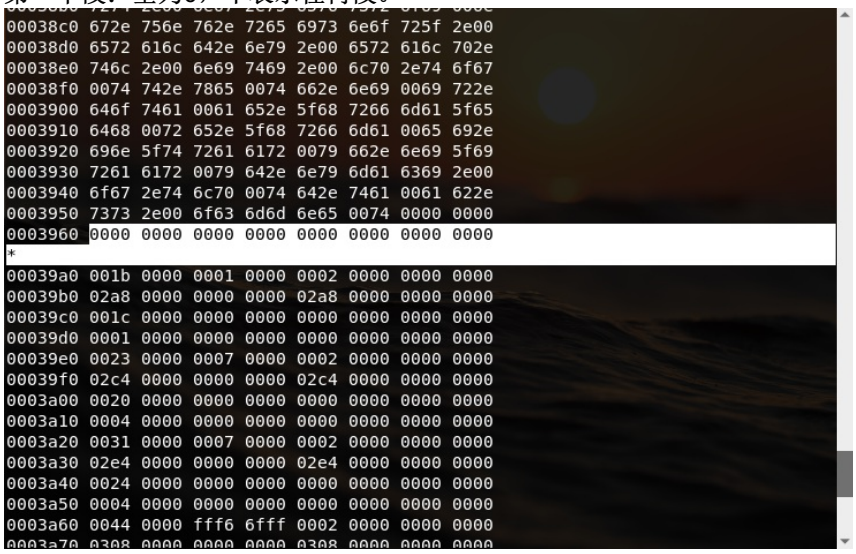
使用`gedit /usr/include/elf.h`查看Section header的结构体。

```
/* Section header.  */

typedef struct
{
  Elf32_Word  sh_name;      /* Section name (string tbl index) */
  Elf32_Word  sh_type;      /* Section type */
  Elf32_Word  sh_flags;     /* Section flags */
  Elf32_Addr  sh_addr;      /* Section virtual addr at execution */
  Elf32_Off   sh_offset;    /* Section file offset */
  Elf32_Word  sh_size;      /* Section size in bytes */
  Elf32_Word  sh_link;      /* Link to another section */
  Elf32_Word  sh_info;      /* Additional section information */
  Elf32_Word  sh_addralign;     /* Section alignment */
  Elf32_Word  sh_entsize;       /* Entry size if section holds table */
} Elf32_Shdr;

typedef struct
{
  Elf64_Word  sh_name;      /* Section name (string tbl index) */
  Elf64_Word  sh_type;      /* Section type */
  Elf64_Xword sh_flags;     /* Section flags */
  Elf64_Addr  sh_addr;      /* Section virtual addr at execution */
  Elf64_Off   sh_offset;    /* Section file offset */
  Elf64_Xword sh_size;      /* Section size in bytes */
  Elf64_Word  sh_link;      /* Link to another section */
  Elf64_Word  sh_info;      /* Additional section information */
  Elf64_Xword sh_addralign;     /* Section alignment */
  Elf64_Xword sh_entsize;       /* Entry size if section holds table */
} Elf64_Shdr;
```

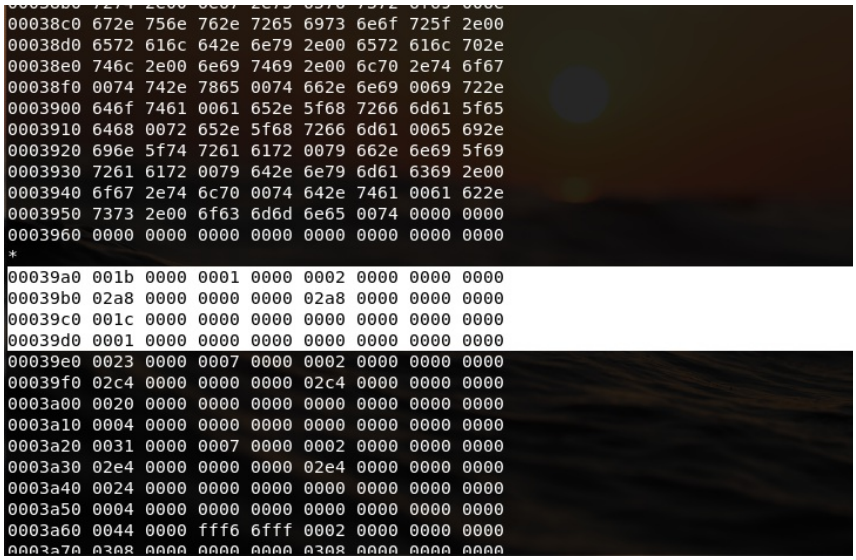
由上面分析可知，section header table中的每个section header所占的size均为64字节，ELF header得到了e_shoff变量的值为0x0000000000006039（不确定对不对），也就是table入口的偏移量，通过看e_shnum的值为0x001e，表示段表入口有30个。

所以从0x0000000000003960（不确定对不对）开始有30个段，每个段占64字节大小，输入`hexdump hello`查看。

- 第一个段：全为0，不表示任何段。

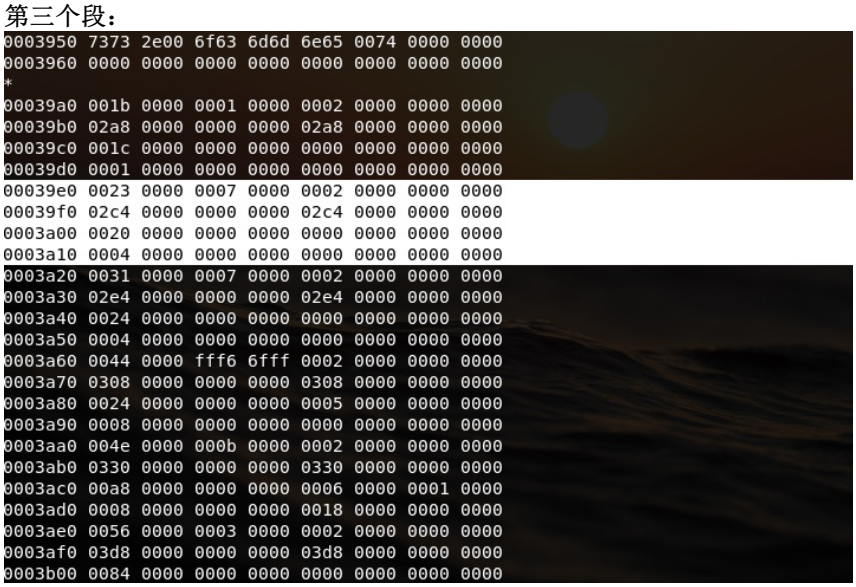


- 第二个段：



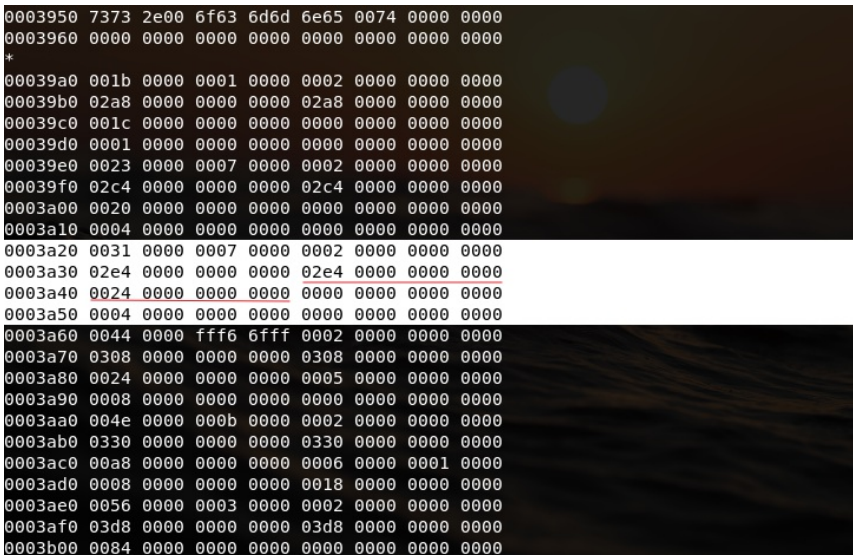
- sh_name（4字节）的值为001b0000表示该段名称在.shstrtab中偏移量，为.inter段。
- sh_type（4字节）的值为00010000表示这个段拥有程序所定义的信息，其格式和含义完全由该程序确定，这里表示PROGBITS。
- sh_flags（8字节）的值为0002000000000000表示alloc和execute。
- sh_addr（8字节）的值为02a8000000000000表示section在内存中的虚拟地址。
- sh_offset（8字节）的值为02a8000000000000(000002a8)表示section与文件头之间的偏移。
- sh_size（8字节）的值为001c000000000000表示文件里面section占用的大小。
- sh_link（4字节）的值为00000000，表示没有链接信息。
- sh_info（4字节）的值为00000000，表示没有辅助信息。
- sh_addralign（8字节）的值为0001000000000000，表示字节对齐长度。
- sh_entsize（8字节）的值为0000000000000000，表示没有入口。

- 第三个段：



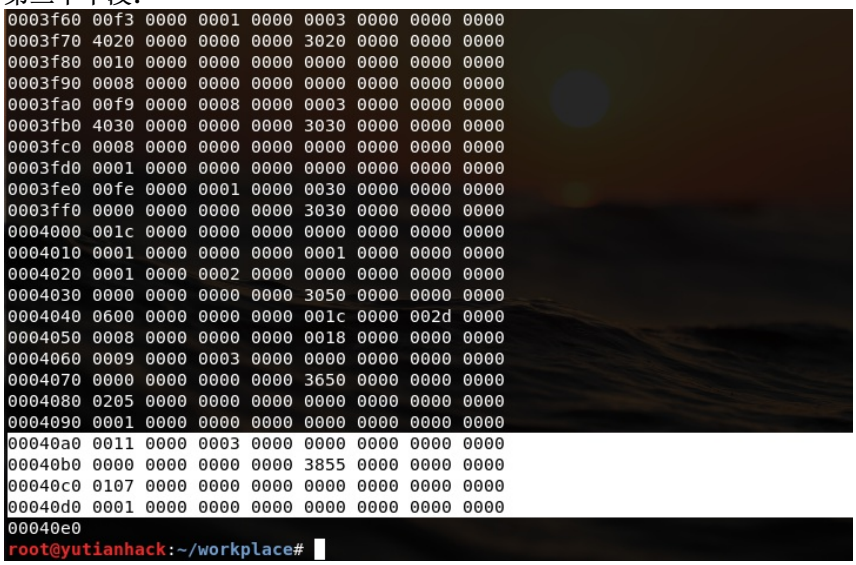
- 段名：note.ABI-tag
- 类型：NOTE
- 标志：（不知道）
- 相对文件头偏移：000002c4
- 占用大小：00000020（不确定）

- 第四个段



- 段名：.note-gnu.build-i
- 类型：NOTE
- 标志：(不知道）
- 相对文件头偏移：000002e4
- 占用大小：00000024（不确定）
  …
  …

- 第三十个段：



- 段名：.shstrtab
- 类型：STRTAB
- 标志：（不知道）
- 相对文件头偏移：00003855

  占用大小：00000107

  ### 参考

  博客1
  博客2
  博客3

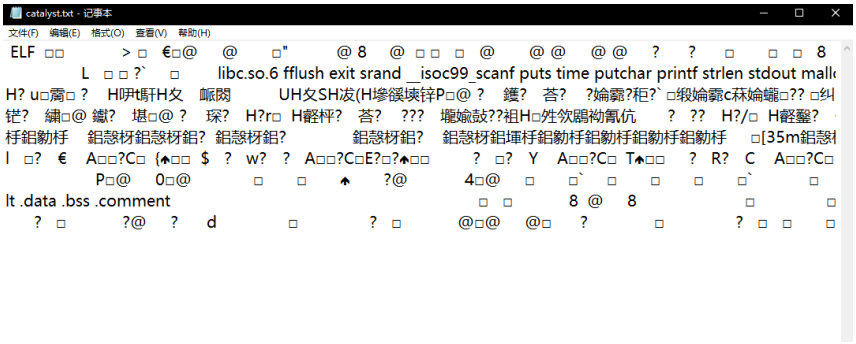
正式开始

题目链接：

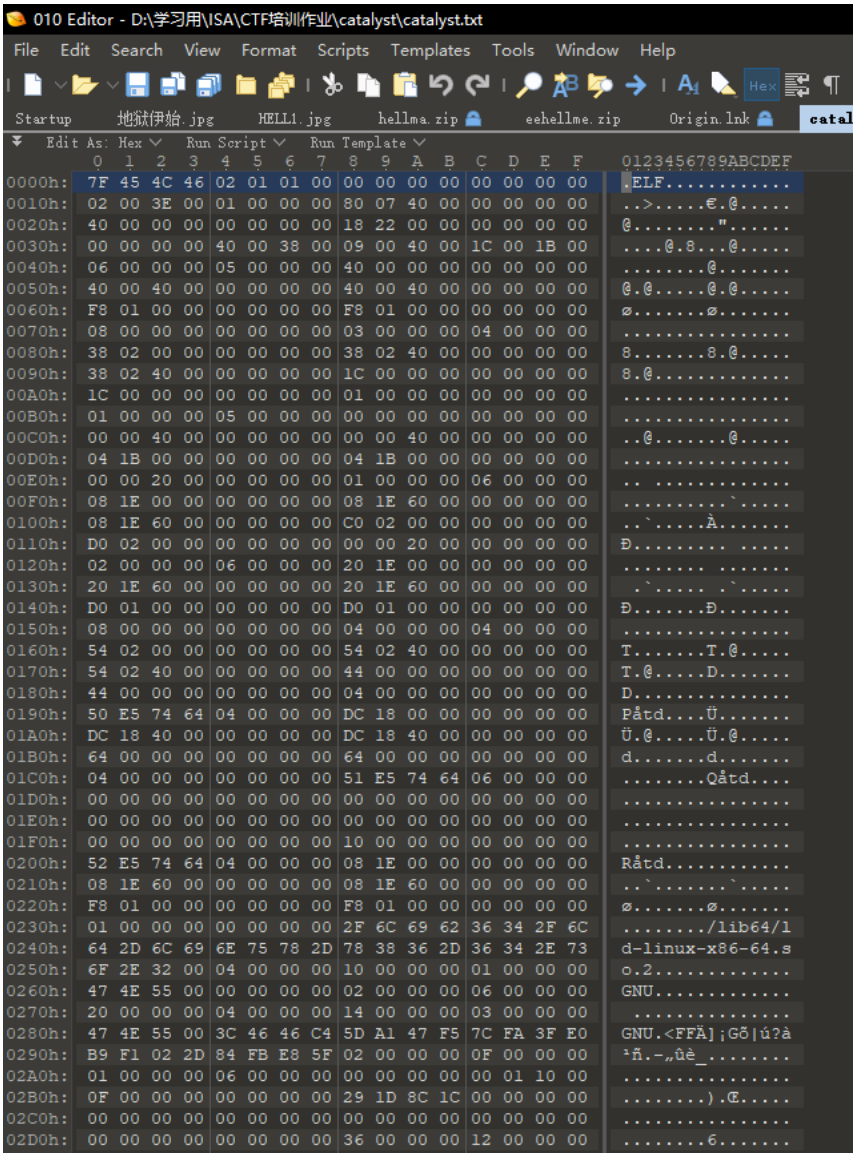
catalyst-system

无能瞎搞

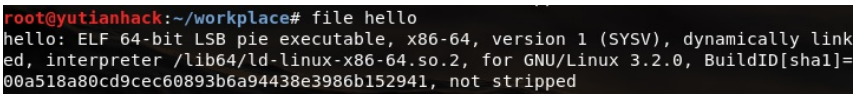
1.预备了这么久真是没想到..一开始下载下来是个文本文件，但是打开是乱码。



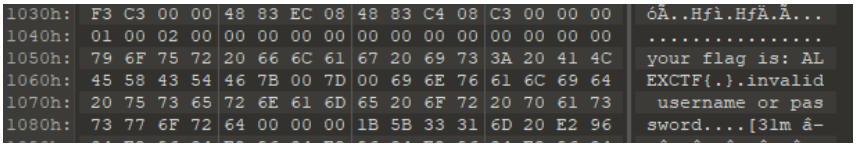
用16进制方式打开后发现开头是.ELF，于是去搜索了，并且按照预备知识学习了一下。

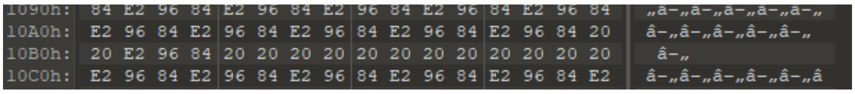


向下划发现这里还有很多有趣的东西

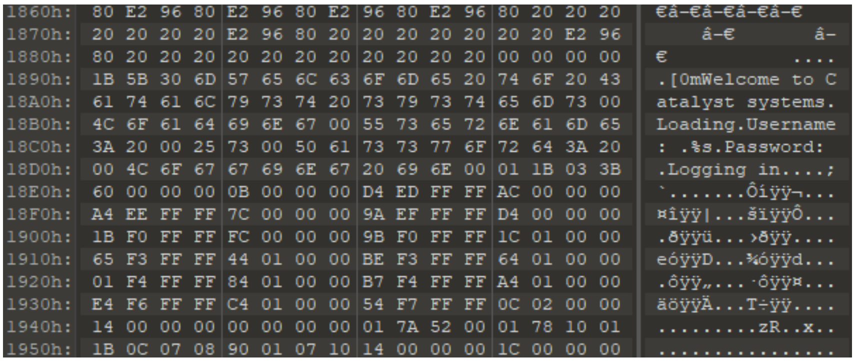


还比如flag

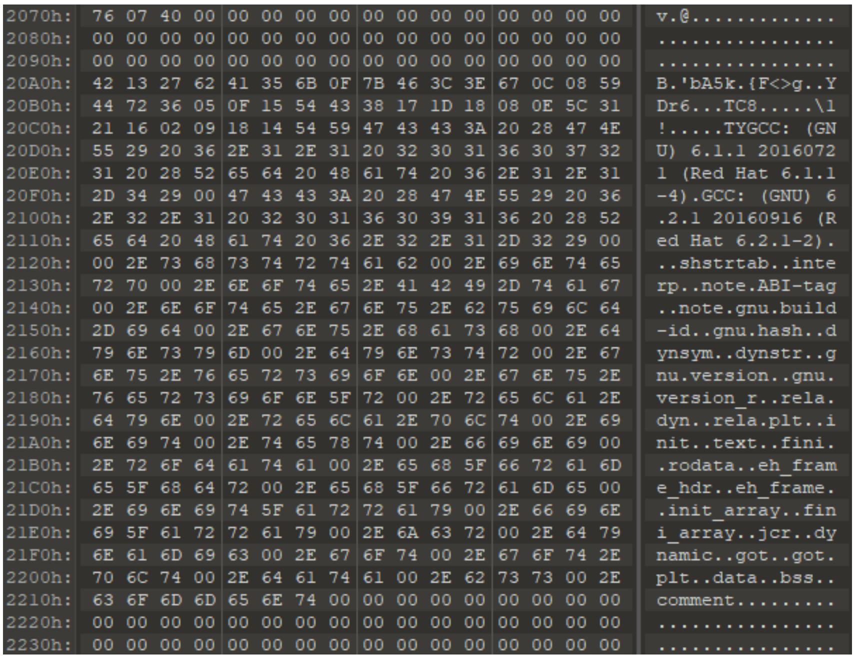




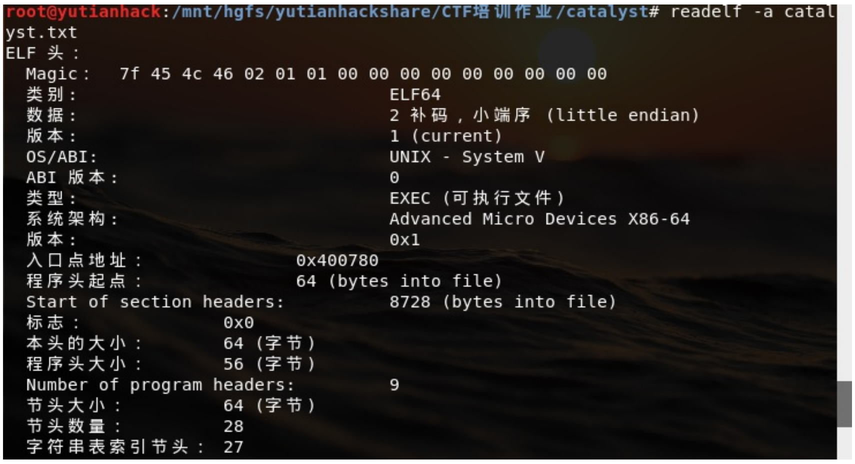
用户名和密码



还有这里很像刚才学的.shstrtab，也像flag的格式

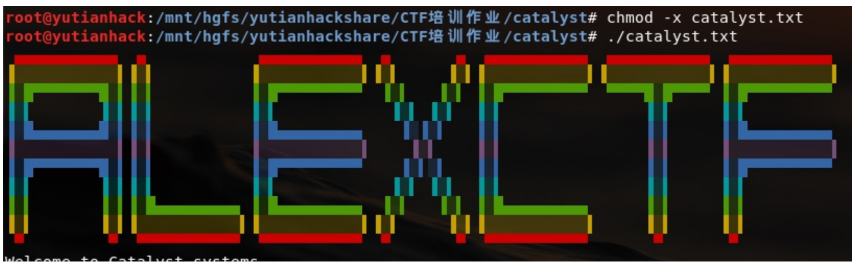


2.啥也不会的我使用预备里学到的指令readelf来查看



发现是一个可执行文件。

3.搜索学习了Linux执行可执行文件的方法，先用`chmod -x catalyst.txt`给执行权限，然后`./catalyst.txt`执行。



```
Welcome to Catalyst Systems
Loading.............
```

Wow，好酷

4.刚才预备里学了一个objdump是反汇编的命令，尝试一下。

```
root@yutianhack:/mnt/hgfs/yutianhackshare/CTF培训作业/catalyst# objdump -x catalyst.txt


catalyst.txt：     文件格式 elf64-x86-64
catalyst.txt
体系结构：i386:x86-64，标志 0x00000112：
EXEC_P, HAS_SYMS, D_PAGED
起始地址 0x0000000000400780

程序头：
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 al
ign 2**3
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r-x
  INTERP off    0x0000000000000238 vaddr 0x0000000000400238 paddr 0x0000000000400238 al
ign 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 al
ign 2**21
         filesz 0x0000000000001b04 memsz 0x0000000000001b04 flags r-x
    LOAD off    0x0000000000001e08 vaddr 0x0000000000601e08 paddr 0x0000000000601e08 al
ign 2**21
         filesz 0x00000000000002c0 memsz 0x00000000000002d0 flags rw-
 DYNAMIC off    0x0000000000001e20 vaddr 0x0000000000601e20 paddr 0x0000000000601e20 al
ign 2**3
         filesz 0x00000000000001d0 memsz 0x00000000000001d0 flags rw-
    NOTE off    0x0000000000000254 vaddr 0x0000000000400254 paddr 0x0000000000400254 al
ign 2**2
         filesz 0x0000000000000044 memsz 0x0000000000000044 flags r--
```

以上我做的这些花里胡哨的东西可以用下图总结

嗯，没有什么用。所以接下来直接寻找write up照着做吧。（顺便这时候我知道了catalyst是催化剂的意思，好浪漫啊。）

IDA启动

值得一提的是，刚刚在执行此文件时很慢，一直卡在“Loading”那里，没有显示出所有我们在16进制中看到的提示字符串“Username:”。

由于以下题解本人还没看懂，暂时止步于此（。。。）（而且已经第四周了。。）

转载于:https://www.cnblogs.com/hardcoreYutian/p/11193928.html