x86_64逆向工程简介

lzhtony / 2018-07-26 11:27:09 / 浏览数 4687 技术文章 技术文章 顶(2) 踩(0)

---

本文翻译自：An Intro to x86_64 Reverse Engineering

系列第二篇：其他练习

---

本文档通过一系列CrackMe程序介绍x86_64二进制逆向工程。逆向工程是了解已编译计算机程序的行为而无需获得其源代码的过程。

关于逆向工程已有很多优秀的教程，但它们主要是在32位x86平台上进行逆向。而现代计算机几乎都是64位的，因此本教程引入了64位的概念。

CrackMe是一类可执行文件，它（通常）由用户输入一个参数，程序对其进行检查，并返回一条消息，告知用户输入是否正确。

如果您喜欢本教程，请考虑支持我的Patreon，这样我就可以更好地做教程。

## 前期准备

### 知识

本教程假定您对编程有一定的了解，但并不需要具备汇编，CPU架构和C编程的知识。您应该知道编译器的功能，但您不必知道如何实现它。同样，您应该知道寄存器是什么

如果您是一个熟练的程序员，但不知道汇编，我建议您看看x86 Crash Course。这是一个10分钟的视频，可以让您了解本教程所需的背景知识。

### CrackMe程序

您可以在GitHub上找到文中讨论的CrackMe程序。克隆这个存储库，并且在不查看源代码的情况下，使用`make crackme01`,`make crackme02`,.....构建所有CrackMe。

### 工具和软件

这些CrackMe仅适用于Unix系统，我使用Linux编写本教程。您需要安装开发环境的基本知识——C编译器（`gcc`）、对象检查工具（`objdump`，`objcopy`，`xxd`）等等。本

```
sudo apt install build-essential gcc xxd binutils
```

您可以在这里安装Radare2。

对于其他系统，通过对应系统的包管理器安装相应的包即可。

## CrackMe解答

注意：在后面的解答中，我会讨论文件偏移。这些值在您的机器上可能会有所不同，但我一定会解释我是如何得到它们的。所以如果您对于某些偏移的值感到困惑，您只

### crackme01.c

`crackme01.64`是一个相对简单的程序。运行后会显示如下输出

```
$ ./crackme01.64
Need exactly one argument.
```

随便给它一个参数，这里用了`lmao`：

```
$ ./crackme01.64 lmao
No, lmao is not correct.
```

这是预料之中的，我们不知道密码。当遇到这种情况时，我们应该首先考虑程序做了些什么。检查字符串是否正确的最简单方法是，将它与存储在二进制文件中的另一个字符

亲自尝试一下：用`cat`、`less`或者其他其他您喜欢的文本编辑器查看可执行程序。

如果我们只是简单地`cat`，我们会得到一堆乱码。有一个名为`strings`的标准Unix工具，它会尝试在给定文件中提取所有有效的字符串（字符串是可打印字符和空字符的组合

```
$ strings ./crackme01.64
/lib/ld-linux.so.2
WXZd
libc.so.6
_IO_stdin_used
```

```
__printf_chk
puts
__cxa_finalize
__libc_start_main
_ITM_deregisterTMCloneTable
__gmon_start__
_Jv_RegisterClasses
_ITM_registerTMCloneTable
GLIBC_2.3.4


...

.dynamic
.data
.bss
.comment
.debug_aranges
.debug_info
.debug_abbrev
.debug_line
.debug_str
.debug_loc
```

这里产生了很多输出。我们可以从中找到一些有用的东西，现在我们只是寻找密码。

亲自尝试一下：在`strings`的输出中寻找密码。这是解决这个问题仅需的方法。

解答

这个问题中，您只需要滚动列表，然后就能发现下面几行：

```
...
[^_]
Need exactly one argument.
password1
No, %s is not correct.
Yes, %s is correct!
;*2$"
...
```

您可以看到我们已经知道的两个字符串：`Need exactly one argument.`和`No, %s is not correct.`.请注意，`%s`是告诉C的`printf`函数打印字符串的控制字符串，并可以猜测最后会替换为我们在命令行输入的字符串。

在这两个字符串之间，我们发现有一个可疑的东西。来试试看：

```
$ ./crackme01.64 password1
Yes, password1 is correct!
```

成功了！您可能会惊讶于在二进制文件上简单地调用`strings`会产生这么多有用的知识。

练习：有一个名为`crackme01e.c`的文件可以使用相同的方法解决。编译并尝试解决它，巩固您的技能。

## crackme02.c

这个 CrackMe 稍微更难一些。您可以尝试上面的步骤，但会发现找到的密码是无效的！

亲自尝试一下：在接着阅读之前，试着想想为什么会这样。

我们用`objdump`来查看程序的实际行为。`objdump`是一个非常强大的二进制文件检查工具，您可能需要使用系统的包管理器进行安装。

二进制程序是一系列机器指令。`objdump`允许我们反汇编这些机器指令，并将它们表示为稍微更易读的汇编助记符。

在这个题目中，运行`objdump -d crackme02.64 -Mintel | less`，我们将获得一个汇编指令清单。我通过`less`管道查看，因为它很长。

第一行告诉我们我们正在看什么：`crackme02.64: file format elf64-x86-64`。它是 Intel x86_64（即AMD64）CPU 架构上的64位 ELF 可执行文件。在这之后有许多节（section），如下所示：

```
Disassembly of section .init:

0000000000000590 <_init>:
```

```
590:   48 83 ec 08             sub    rsp,0x8
594:   48 8b 05 3d 0a 20 00    mov    rax,QWORD PTR [rip+0x200a3d]        # 200fd8 <__gmon_start__>
59b:   48 85 c0                test   rax,rax
59e:   74 02                   je     5a2 <_init+0x12>
5a0:   ff d0                   call   rax
5a2:   48 83 c4 08             add    rsp,0x8
5a6:   c3                      ret
...
```

其中大多数的节是在编译后由链接器插入的，因此与检查密码的算法无关。我们可以跳过除.text节之外的所有内容。它开始是这样的：

```
Disassembly of section .text:

00000000000005e0 <_start>:
5e0:   31 ed                   xor    ebp,ebp
5e2:   49 89 d1                mov    r9,rdx
5e5:   5e                      pop    rsi
5e6:   48 89 e2                mov    rdx,rsp
5e9:   48 83 e4 f0             and    rsp,0xfffffffffffffff0
5ed:   50                      push   rax
5ee:   54                      push   rsp
...
```

同样，这是链接器插入的函数。我们不关心任何与main函数无关的事情，所以继续滚动直到您看到：

```
0000000000000710
:
710:   48 83 ec 08             sub    rsp,0x8
714:   83 ff 02                cmp    edi,0x2
717:   75 68                   jne    781
719:   48 8b 56 08             mov    rdx,QWORD PTR [rsi+0x8]
71d:   0f b6 02                movzx  eax,BYTE PTR [rdx]
720:   84 c0                   test   al,al
...
```

在最左的一列中列出了每个指令的地址（十六进制）。往右一列是原始机器代码字节，表示为十六进制数对（两个十六进制数组成一组）。最后一列是 objdump 生成的等效汇编代码。

我们分解这个程序。首先是 sub rsp，0x8
，这将堆栈指针向下移动8，在堆栈上为8个字节的变量分配空间。请注意，我们对这些变量一无所知。这些空间可以表示8个字符，也可以是一个指针（它是64位可执行文件

接下来，有一个非常标准的 jump-if 条件：

```
cmp    edi,0x2
jne    781
```

如果您不知道这些指令的作用，可以去搜索。在这里，我们将edi寄存器与十六进制数2进行比较（cmp），如果它们不相等则跳转（jne）。

所以问题是，那个寄存器中存放了什么？这是一个Linux x86_64可执行文件，因此我们可以查找调用约定（Wikipedia）。发现edi是目标索引（Destination Index）寄存器的低32位，是函数的第一个参数存放的位置。想想main函数是如何用C编写的，它的声明是：`int main■int argc■char ** argv■`。所以这个寄存器保存第一个参数：argc，就是程序的参数个数。

查找明文字符串

因此，这个比较跳转是检查程序是否有两个参数。（注意：第一个参数是程序的名称，所以它实际上检查是否有一个用户提供的参数。）如果不是，它会跳转到主程序的另一

```
lea    rdi,[rip+0xbc]
call   5c0 <.plt.got>
mov    eax,0xffffffff
jmp    77c
```

在这里，我们将一个值的地址加载（lea）到rdi中（还记得吗，这是函数的第一个参数），然后调用一个地址是5c0的函数。看一下该行的反汇编：

```
5c0: ff 25 02 0a 20 00 jmp QWORD PTR [rip+0x200a02] # 200fc8
```

objdump注释了这条指令，告诉我们它正在跳转到libc函数puts。该函数只需要一个参数：一个指向字符串的指针，然后将其打印到控制台。所以这段代码打印了一个字符

要回答这个问题，我们需要查看载入到rdi中的内容。看看这条指令：`lea rdi,[rip + 0xbc]`。这计算了指令指针（Instruction Pointer，指向下一条指令的指针）向前0xbc的地址，并将该地址存储在rdi中。

因此我们打印的是在此指令之前的0xbc字节中的内容。我们可以自己计算：0x788（下一条指令）+ 0xbc（偏移）= 0x845。

我们可以使用另一个标准Unix二进制工具来查看特定偏移量的原始数据：xxd。这个题目中，执行xxd -s 0x844 -l 0x40 crackme02.64。其中，-s是表示跳到（skip）指定位置，使输出从我们感兴趣的偏移开始。-l是指输出长度（length），使输出只有0x40个字符长，而不是整个文件的余

```
$ xxd -s 0x844 -l 0x40 crackme02.64
00000844: 4e65 6564 2065 7861 6374 6c79 206f 6e65  Need exactly one
00000854: 2061 7267 756d 656e 742e 004e 6f2c 2025   argument..No, %
00000864: 7320 6973 206e 6f74 2063 6f72 7265 6374  s is not correct
00000874: 2e0a 0070 6173 7377 6f72 6431 0059 6573  ...password1.Yes
```

所以现在我们知道这段代码打印一个字符串"Need exactly one argument."这就是当您指定太多或太少的参数时，您会看到的程序行为。

基础流分析

这段代码最重要的部分是最后的无条件跳转，它转到地址77c：

```
add    rsp,0x8
ret
```

这段代码从堆栈中删除局部变量并返回，仅此而已。如果没有为二进制文件提供正好2个参数——它自己的名称和一个命令行参数——它就会退出。

我们可以用C代码编写这个程序：

```
int main(int argc, char** argv){
   if (argc != 2) {
       puts("Need exactly one argument.");
       return -1;
   }

   // ■■■■■■■■■■
}
```

为了找出程序接下来的部分中发生了什么神奇的事情，我们需要查看程序的流程。假设argc检查通过（不进行0x717的跳转），程序将进入该块执行：

```
mov    rdx,QWORD PTR [rsi+0x8]
movzx  eax,BYTE PTR [rdx]
test   al,al
je     761
```

第一条指令将地址[rsi + 0x8]的四字（64位值）移入rdx。什么是 rsi ？完整64位源索引寄存器（the full 64-bit Source Index register）？实际上这是Linux x86_64调用约定中的第二个参数。 所以在C语言中，这是argv + 8的值，或者argv[1]，因为argv的类型是char **。

下一条指令移动存储在rdx中的地址上的一个字节并向高位填充零（movzx）。换句话说，移动了*argv[1]，或argv[1][0]。现在eax寄存器（The Accumulator register）除了最后8位为argv[1]（即程序的命令行参数）的第一个字节，高位全为零。

test al■al相当于cmp al,0。al是累加器寄存器的低8位。这个程序块相当于C代码：

```
if (argv[1][0] == 0) {
   // do something
}
```

那么地址0x761中是什么？它是这样的：

```
lea    rsi,[rip+0x119]        # 881 <_IO_stdin_used+0x41>
mov    edi,0x1
mov    eax,0x0
call   5c8 <.plt.got+0x8>
mov    eax,0x0
add    rsp,0x8
ret
```

逆向工程师最重要的技能之一是注意到代码的模式，您在这里就可以看到一个。这里，程序通过lea复制了一个指令指针的相对偏移量到rsi，然后调用一个函数。

使用和上面相同技术，可以知道这个函数是printf。printf的参数是一个格式字符串和可变数量的参数。所有可变函数都需要使用eax累加寄存器来保存一个值，告诉程序 eax■0x0指令中看到的那样）。rdx寄存器已经存放了指针argv[1]，所以这是第二个命令行参数。

那格式字符串是什么？我们使用与以前相同的技术，但这次我没有把objdump添加的注释去掉，它帮我们做了数学运算。

所以运行xxd -s 0x881 -l 0x40 crackme02.64，得到这里的格式字符串是Yes, %s is correct!。看起来很好！另外我们可以看到，在函数调用之后（在地址0x77c，这是一个很有用的地址，要记住），局部变量的空间从堆栈中删除，函数返回。返回值总是

所以我们的C代码看起来像这样：

```
int main(int argc, char** argv){
   if (argc != 2) {
       puts("Need exactly one argument.");
       return -1;
   }

   if (argv[1][0] == 0) {
       printf("Yes, %s is correct.", argv[1]);
   }

   // ■■■■■■■■■■
}
```

我们所要做的只是提供一个字符串，其第一个字节为0——也就是空字符串：

```
$ ./crackme02.64 ""
Yes,  is correct!
```

从某种意义上说，我们已经完成了这个CrackMe，但是我们继续看看接下来的代码。

如果检查失败，则代码转到这里（地址0x724处）：

```
cmp    al,0x6f
jne    794
```

回想一下，由于我们假设检查成功的跳转没有执行，所以al中现在存放着`argv[1][0]`。这段代码检查它是否不等于0x6f（十进制111; ASCII字符'o'）。如果是就跳转到地址0x794。

```
lea    rsi,[rip+0xc4]        # 85f <_IO_stdin_used+0x1f>
mov    edi,0x1
mov    eax,0x0
call   5c8 <.plt.got+0x8>
mov    eax,0x1
jmp    77c
```

这又是一个打印并返回的代码块。最后无条件跳转（`jmp`）到0x77c，程序删除其局部变量的堆栈空间并返回。

这个代码块不是打印成功消息，而是打印"No, %s is not correct."，格式化字符串填入命令行参数，然后返回失败代码1。那我们就知道正确的消息以字母"o"开头，如果不是就会判定失败。

```
int main(int argc, char** argv){
   if (argc != 2) {
       puts("Need exactly one argument.");
       return -1;
   }

   if (argv[1][0] == 0) {
       printf("Yes, %s is correct.", argv[1]);
       return 0;
   }

   if (argv[1][0] != 'o') {
       printf("No, %s is not correct.", argv[1]);
       return 1;
   }

   // ■■■■■■■■■■
}
```

假设跳转不发生，那么我们来到地址0x728的代码块处：

```
mov    esi,0x1
mov    eax,0x61
mov    ecx,0x1
lea    rdi,[rip+0x139]        # 877 <_IO_stdin_used+0x37>
movzx  ecx,BYTE PTR [rdx+rcx*1]
test   cl,cl
je     761
```

在这里，我们给寄存器加载一些常量，然后将一个指针加载到`rdi`中。这个指针指向字符串"password1"，但我们知道这不是正确的密码。究竟发生什么了？

下一条指令移动一个地址在rdx + rcx的字节。rdx里面是什么？我们向上翻一翻，到0x719的代码处，我们看到它加载了rsi + 0x8的值，也就是argv[1]。所以这里其实是在索引那个字符串，ecx = argv[1][1]。

之前说过，逆向工程最重要的技能是识别代码的模式。这是我们在上面已经见过的汇编片段：寄存器test自己，紧接着je，等价于"如果寄存器为零则跳转"。

所以，如果在argv[1][1]处是一个零字节，那么就跳转到0x761。那里的代码逻辑是什么？这是我们刚刚逆向过的一个代码块，它打印成功字符串并退出，返回码为0。伪

```
int main(int argc, char** argv){
   if (argc != 2) {
       puts("Need exactly one argument.");
       return -1;
   }

   if (argv[1][0] == 0 || argv[1][1] == 0) {
       printf("Yes, %s is correct.", argv[1]);
       return 0;
   }

   if (argv[1][0] != 'o') {
       printf("No, %s is not correct.", argv[1]);
       return 1;
   }

   // ■■■■■■■■■■
}
```

如果第二个字符不是零，会怎样呢？继续向下，看0x746处的代码：

```
movsx  eax,al
sub    eax,0x1
movsx  ecx,cl
cmp    eax,ecx
jne    794
```

这里我们将eax除最低8位之外都清零，并减去1。然后同样将ecx除最低8位之外都清零，并将eax与ecx进行比较。如果它们不相等，就跳转到0x794。这是又一个我们已经

这个代码是实现什么的？从上面我们可以知道，eax包含一个字节0x61（十进制97，ASCII字符'a'）。它减去1，是0x60（十进制96，ASCII字符'`'）。所以我们就知道了，
我们的伪代码如下：

```
int main(int argc, char** argv){
   if (argc != 2) {
       puts("Need exactly one argument.");
       return -1;
   }

   if (argv[1][0] == 0 || argv[1][1] == 0) {
       printf("Yes, %s is correct.", argv[1]);
       return 0;
   }

   if (argv[1][0] != 'o' || argv[1][1] != 0x60) {
       printf("No, %s is not correct.", argv[1]);
       return 1;
   }

   // ■■■■■■■■■■
}
```

如果它们相等，那么就到了地址0x753的代码处：

```
add    esi,0x1
movsxd rcx,esi
movzx  eax,BYTE PTR [rdi+rcx*1]
test   al,al
jne    73e
```

一开始程序使esi加一。（esi在前一个块中赋值为1。）然后将该值移动到rcx的低32位。

然后，程序从rdi + rcx加载一个字节。rdi是argv[1]，rcx是esi + 1（此时为2）。所以这里的程序加载了argv[1][2]。更准确地说，它加载了argv[1][rcx]（您稍后会明白为什么这一点很重要）。

然后代码检查它是否等于0，如果不是就跳转到0x73e：

```
movzx  ecx,BYTE PTR [rdx+rcx*1]
test   cl,cl
je     761
```

我们之前见过这个代码块，这是上面几节见过的检查代码。它从argv[1][ecx]加载一个字节并检查它是否为零，如果是，它会跳转到判定成功的代码块，如果不是，它会

现在我们已经发现了整个循环，我们看看它的所有指令，从0x73e开始到0x75f结束。

回想一下，几个块之前，rdi加载了字符串password1的地址，但这不是正确的密码。在这里我们可以发现原因。从这个字符串加载的字节，在将它们与实际输入进行比较

```
movzx  ecx,BYTE PTR [rdx+rcx*1] ; load a byte from argv[1]
test   cl,cl                    ; check if that byte is zero
je     761           ; if so, jump to success
movsx  eax,al
sub    eax,0x1                  ; decrement comparison byte
movsx  ecx,cl
cmp    eax,ecx                  ; check if the correct byte == the input byte
jne    794           ; if it doesn't match, jump to failure
add    esi,0x1                  ; increment index into comparison string
movsxd rcx,esi                  ; place that index in CX
movzx  eax,BYTE PTR [rdi+rcx*1] ; load the next byte from the comparison string
test   al,al                    ; Check that that byte isn't zero
jne    73e           ; If it's not zero, loop
```

虽然我们会像下面的C代码那样编写它，但编译器实际上将循环检查的第二部分移动到循环的末尾，并在那里加载比较字符串的下一个字节。

注意：此代码在本教程的原始版本中不正确。感谢empwill的指正。我提出这一点，是想让读者意识到即使是经验丰富的逆向工程师也会犯错误，而且这些错误可以预见和修

亲自尝试一下：要找到正确的密码，看下面这个C代码就足够了。试试看，去找到密码！

```
int main(int argc, char** argv){
   if (argc != 2) {
       puts("Need exactly one argument.");
       return -1;
   }
   // This pointer is in rdi in our disassembled binary
   char* comparison = "password1";
   // This is the value used to index the argv[1]
   int i = 0;

   while (argv[1][i] != 0 && (comparison[i]) != 0) {
       if (argv[1][i] != comparison[i] - 1) {
           printf("No, %s is not correct.", argv[1]);
           return 1;
       }
       i++;
   }

   printf("Yes, %s is correct.", argv[1]);
   return 0;
}
```

确实只要这个代码就足够了。只要简单地对password1字符串中的每个字符减去1，就得到“o`rrvnqc0”。试试吧：

```
$ ./crackme02.64 o\`rrvnqc0
Yes, o`rrvnqc0 is correct!
```

您可能已经敏锐地觉察到这个二进制文件存在问题，它会接受这些字符串中的任何一个：o，o`，o`r，o`rr等等都会生效！显然这个方法用于您的产品密钥中不是很好。此外
""）。

如果您读到这里，那就恭喜您！逆向工程很难，但些这是它的核心部分，而且从此以后它会变得更加容易。

练习：有一个名为crackme02e.c的文件可以使用相同的方法解决。编译并尝试解决它，巩固您的技能。

## crackme03.c

下一个CrackMe会稍微难一些。在crackme02中，我们人为查看每个分支，在心里构建了整个执行流程。随着程序变得更复杂，这种方法就变得不可行了。

### Radare 分析工具

不过逆向工程社区有很多聪明人，并且开发出很多好工具可以自动完成大量的分析。其中一些如Ida Pro，售价高达5000美元。我个人最喜欢的是Radare2（Random data recovery），它完全免费且开源。

运行crackme03.64，我们可以看到它的行为与前两个题目基本上相同。它需要且只需要一个参数，当我们提供一个参数时，它会告诉我们这是错误的，这很有用。

这一次，我们使用radare2（或r2命令）打开它，而不用objdump：r2 ./crackme03.64。这时您会看到一个提示符界面。输入"?"能看到帮助信息。Radare是一个非常强大的工具，但对于这个题目，我们不需要用到它太多功能。在下面这个

```
[0x000005e0]> ?
Usage: [.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
| a[?]                    Analysis commands
| p[?] [len]              Print current block with format and length
| s[?] [addr]             Seek to address (also for '0x', '0x1' == 's 0x1')
| V                       Enter visual mode (V! = panels, VV = fcngraph, VVV = callgraph)
```

需要注意的一点是Radare自带文档。如果您想知道一个命令是什么用的，只需在它之后输入一个问号"?"。例如我们想分析当前的程序：

```
[0x000005e0]> a?
|Usage: a[abdefFghoprxstc] [...]
| ab [hexpairs]    analyze bytes
| aa[?]            analyze all (fcns + bbs) (aa0 to avoid sub renaming)
| ac[?] [cycles]   analyze which op could be executed in [cycles]
| ad[?]            analyze data trampoline (wip)
| ad [from] [to]   analyze data pointers to (from-to)
| ae[?] [expr]     analyze opcode eval expression (see ao)
| af[?]            analyze Functions
| aF               same as above, but using anal.depth=1
| ag[?] [options]  output Graphviz code
| ah[?]            analysis hints (force opcode size, ...)
| ai [addr]        address information (show perms, stack, heap, ...)
| ao[?] [len]      analyze Opcodes (or emulate it)
| aO               Analyze N instructions in M bytes
| ar[?]            like 'dr' but for the esil vm. (registers)
| ap               find prelude for current offset
| ax[?]            manage refs/xrefs (see also afx?)
| as[?] [num]      analyze syscall using dbg.reg
| at[?] [.]        analyze execution traces
Examples:
f ts @ `S*~text:0[3]`; f t @ section..text
f ds @ `S*~data:0[3]`; f d @ section..data
.ad t t+ts @ d:ds
```

亲自尝试一下：翻阅一下帮助，通过Google查询您不知道的术语。在这篇文章里不会涉及其中很多很酷的功能，但这会激发您进行一些尝试。

## 自动化分析

我们可以用它的命令aaaa：使用所有正常及实验中技术分析函数。

这样Radare会给我们返回一个函数列表。我们可以用afl查看它：分析函数，显示列表（analyze functions, displaying a list）。

```
[0x000005e0]> afl
0x00000000    3 73   -> 75   fcn.rsp
0x00000049    1 219          fcn.00000049
0x00000590    3 23           sym._init
0x000005c0    1 8            sym.imp.puts
0x000005c8    1 8            sym.imp.__printf_chk
0x000005d0    1 16           sym.imp.__cxa_finalize
0x000005e0    1 43           entry0
0x00000610    4 50   -> 44   sym.deregister_tm_clones
0x00000650    4 66   -> 57   sym.register_tm_clones
0x000006a0    5 50           sym.__do_global_dtors_aux
0x000006e0    4 48   -> 42   entry1.init
0x00000710    7 58           sym.check_pw
0x0000074a    7 203          main
0x00000820    4 101          sym.__libc_csu_init
0x00000890    1 2            sym.__libc_csu_fini
0x00000894    1 9            sym._fini
```

我们只要关心main和check_pw两个函数。

亲自尝试一下：想想看我为什么可以立即判断出其他函数是无用的，善用搜索引擎。

通过`pdf@main`指令，Radare可以为我们反汇编一个函数：打印main函数的反汇编（print disassembly of a function @ (at) the symbol called `main`）。Radare还支持通过Tab进行上下文自动补全。例如，如果您输入`pdf@sym`，并按Tab键，您将获得符号表中所有函数的列表。

总之，首先要注意的是Radare会对反汇编结果进行语法高亮，添加大量注释，甚至命名一些变量。它也做了一些分析来确定变量的类型。在这个题目中，我们有9个本地堆栈

程序的开头我们非常熟悉。从0x74a开始：

```
push rbx
sub rsp, 0x10
cmp edi, 2
jne 0x7cc
```

我们可以发现函数首先为局部变量分配16个字节的内存，然后是一个`if`语句。回想一下，DI寄存器保存了函数的第一个参数。因为这是`main`函数的参数，所以该参数是`arg`
`(argc != 2) jump somewhere`。

在Radare中，查看`jne`指令的左侧，您会看到一条箭头从该指令出发，并向下指向到0x7cc，我们可以看到：

```
lea rdi, str.Need_exactly_one_argument. ; 0x8a4 ; "Need exactly one argument." ; const char * s
call sym.imp.puts           ; int puts(const char *s)
mov eax, 0xffffffff         ; -1
jmp 0x7c6
```

还记得在我们的二进制文件中搜索字符串有多麻烦吗？Radare为我们做了这些：为我们提供了地址，方便的别名以及字符串文字的值。它还分析出被调用的函数，这非常方
exactly one argument."

然后它给`eax`装入-1并跳转到0x7c6。我们可以通过箭头（或者通过滚动并寻找地址）来查看它，但还有一种更有趣的方式。

可视化流程分析

Radare提供了一种称为"可视化模式"的功能。我们需要先把Radare的内部光标移动到我们想要分析的函数，使用`seek`命令：`s`
`main`。您会注意到提示符从`[0x000005e0]>`更改为`[0x0000074a]>`，表示当前位置已移至`main`函数中的第一条指令，然后输入`VV`（可视模式2）。这时您应该会看到包含

每当出现跳转指令时，代码块就结束了，并且出现指向其他块的箭头。例如，在顶部块（函数的开头）中，检查命令行参数个数的`jne`指令引出一红一绿两个箭头。

在右边您会看到一个类似这样的块：

```
.-------------------------------------------.
|  0x7cc ;[ga]                              |
|      ; const char * s                     |
|      ; 0x8a4                              |
|      ; "Need exactly one argument."       |
| lea rdi, str.Need_exactly_one_argument.   |
| call sym.imp.puts;[gh]                    |
|    ; -1                                   |
| mov eax, 0xffffffff                       |
| jmp 0x7c6;[gg]                            |
`-------------------------------------------'
```

这就是我们刚刚分析的块。使用键盘方向键键跟随蓝色（无条件）箭头向下看看这个块之后会发生什么。您会在底部看到一个0x7c6的块，这个块可以从程序中的许多位置无

```
add rsp, 0x10
pop rbx
ret
```

这里释放堆栈空间并返回。所以这个程序的行为与我们看过的其他程序一样：如果没有正确数量的参数，它会打印一个字符串并退出，返回错误代码（`eax`加载了-1）。

亲自尝试一下：在控制流程图中查看程序的其余部分，找到打印失败消息的块，有两个判断可以通向那里。您能弄清楚它们做了什么吗？
回想一下，`test eax,eax`紧接着`je`表示"如果`eax`为零则跳转"。x86指令集有详细的文档，如果您不知道指令的作用，请查阅！

如果我们从第一个块向下进入没有执行`jne`的红色分支（即正好有2个字符串传递给二进制文件），您将看到在0x754的这些指令：

```
mov dword [local_9h], 0x426d416c ; [0x426d416c:4]=-1
mov word [local_dh], 0x4164 ; [0x4164:2]=0xffff
mov byte [local_fh], 0
mov word [local_6h], 0
mov byte [local_8h], 0
mov byte [local_2h], 2
mov byte [local_3h], 3
mov byte [local_4h], 2
mov byte [local_5h], 3
```

```
mov byte [local_6h], 5
mov rbx, qword [rsi + 8]    ; [0x8:8]=0
mov eax, 0
mov rcx, 0xffffffffffffffff
mov rdi, rbx
repne scasb al, byte [rdi]
cmp rcx, 0xfffffffffffffff8
je 0x7df
```

这个块一大部分的工作是将一堆值加载到内存中。这里Radare不是显示实际地址，而是根据其堆栈偏移命名每个局部变量。向上滚动到最开始的块，我们可以看到local_2

在把这些值加载到局部变量之后，它将地址rsi + 8的内存加载到rbx中。回想一下x86_64调用约定，rsi是第二个命令行参数：argv。所以rsi + 8是argv[1]。然后它给rax载入0，rcx载入0xffffffffffffffff，rdi载入rbx的值，该值刚刚从argv[1]得到。

然后它运行repne scasb指令。这是x86的一个奇怪但快速的指令：它是一个获得字符串长度的原生指令。repne表示在不相等时重复执行（repeat while not equal），scasb表示按字节扫描和比较——有关详细信息，请参阅此处。

因此，该指令将各字节与al的值（此处为0）依次进行比较，从rdi中的存储器地址开始，并对rdi进行累加，同时从rcx中减去1（rcx中的"C"是指计数counter寄存器）。

不管怎样，一旦完成repne scasb操作，rcx将存储着0xffffffffffffffff减去字符串的长度。我们可以看到下一条指令将它与0xfffffffffffffff8进行比较。因此，如果字符串长度是0xffffffffffffffff - 0xfffffffffffffff8 = 7字节（包括终止字符），则跳转，否则不跳转。

如果不进行跳转，则流程进入到0x7a8处的块，会打印失败字符串。因此，我们可以确定正确的密码恰好是6个字节（要去掉终止符）。

函数

更有趣的是进行跳转的部分。

```
lea rdx, [local_2h]
lea rsi, [local_9h]
mov rdi, rbx
call sym.check_pw
test eax, eax
je 0x7a8
```

程序加载一些局部变量的地址，还有argv[1]（记得吗？它被存在rbx中），然后调用一个函数：sym.check_pw。当然，二进制文件中只存有函数的偏移量，但Radare可 eax,eax表示如果eax为零，则执行je跳转）。

那么这个函数到底是做什么的呢？
先回想一下x86_64调用约定。rdi，rsi和rdx（在调用之前赋值的三个寄存器）是函数的前三个参数。所以在C中，调用看起来像这样：

```
int result = check_pw(argv[1], &local_9h, &local_2h);
if (result == 0) {
   // ■■
} else {
   // ■■
}
```

那么问题就转换为check_pw究竟做了什么？为了弄明白这个，我们需要退出视觉模式（连按两次q），并进入这个函数（s sym.check_pw），然后查看流程图（VV）。

很明显，这个函数包含一个循环。main函数里无论怎么跳转流程都会一直向下进行，而在check_pw中，靠近底部的一个块有一个跳到顶部的jne指令。再仔细一点看，我们

这种高级分析只能通过流程图进行，并且这是使用Radare等工具的主要优势之一。刚接触逆向工程时，我亲手绘制流程图，是因为我不知道这些免费工具的存在。不要那样

这个函数首先赋值一个64位通用寄存器r8d，其值为0。然后跳转到下一个块（0x716）：

```
movsxd rax, r8d
movzx ecx, byte [rdx + rax]
add cl, byte [rsi + rax]
cmp cl, byte [rdi + rax]
jne 0x73e;[gb]
```

这个块将r8d（其中是零）赋值给rax，然后从函数的第三个参数加载一个字节，由eax索引。回到我们的参数列表，这个参数是&local_2h，所以它加载了(&local_2h)

然后程序把它与用eax索引的第二个参数中的一个字节（(&local_9h)[0]）相加，并将起与用eax索引的第一个参数中的一个字节（argv[1][0]）进行比较。注意这是一

```
while (/* ■■■■■■ */) {
   char temp  = arg3[eax] + arg2[eax];
   if (temp != arg1[eax]) {
       return 0; // ■■
```

```
   }
}
```

如果跳转不执行，代码会来到0x725处：

```
add r8d, 1
movsxd rax, r8d
cmp byte [rsi + rax], 0
je 0x744;[gd]
```

这里会增加循环计数器，检查用循环计数器索引的第二个参数的那个字节是否为零。如果是，它会跳转到返回成功的代码（0x744）。否则，它继续循环。更新的C代码如下

```
while (arg2[eax] != 0) {
   char temp  = arg3[eax] + arg2[eax];
   if (temp != arg1[eax]) {
       return 0; // ■■
   }

   eax++;
}

return 1;
```

这样就能很容易看出`check_pw`在做什么：它比较两个字符串，但它逐个地修改了其中一个字符串的字符。

看看`main`中传递给函数的参数，我们可以看到这个程序把`(&local_2h)[eax]`和`(&local_9h)[eax]`相加。可以回到`main`函数（退出可视模式，执行`pdf@main`）来查看

这两个变量都在堆栈上。我们之前知道`check_pw`只会在一个含有6个字符的字符串上被调用，因此我们只需要查看6个值。这是`local_2h`之后的值（您可以看到它们在`mai`

我们再看一遍，堆栈变量的赋值从地址0x754开始：

```
mov dword [local_9h], 0x426d416c ; [0x426d416c:4]=-1
mov word [local_dh], 0x4164 ; [0x4164:2]=0xffff
mov byte [local_fh], 0
mov word [local_6h], 0
mov byte [local_8h], 0
mov byte [local_2h], 2
mov byte [local_3h], 3
mov byte [local_4h], 2
mov byte [local_5h], 3
mov byte [local_6h], 5
```

在按顺序将字节大小的值移入`local_2h`到`local_6h`之前，`local_6h`（即`rsp + 0x6`）被载入一个字（word）大小的0（这是Intel语法，所以一个字是16位。请参阅这里）。这就是说`rsp + 0x6`和`rsp + 0x7`都被置为零。

注意，Radare完全没有意识到这些值是在一个数组中，更不要说告诉我们它被初始化了什么值，尽管它完全是静态的数组。这是需要人脑进行逆向工程的一个部分。计算机

总之，我们从`local_2h`开始的值表是`[2,3,2,3,5,0]`。这些不是可打印的ASCII字符，因此硬编码的密码可能存储在另一个参数中：`local_9h`。

最上方的`mov`指令移动了一个双字（dword），这是一个32位的值，接下来是一个字（word）大小的值，然后是一个字节大小的零。这有4 + 2 = 6个字节，加上一个空终止符，所以这三个指令一起组成了一个字符串。 如果我们按字节分隔并写出这些值，则更明显一些：`42 6d 41 6c 41 64 00`。这很明显是以空字符结尾字符串的格式，其值都在可打印的ASCII范围内。

剩下的就是为它们添加偏移量，就得到`44 70 43 6e 44 64 00`。将这些字节转换为ASCII字符，我们得到：`DpCnDd`。

很明显，只要将字符串输入二进制文件……失败了。怎么回事？

> 亲自尝试一下：为什么会这样？这和x86组织数据的方式有关，很基础的知识。

原因是x86处理器是小端序的。也就是在多字节值中需要从右到左读取字节，而不是从左到右。只需翻转`local_9h`和`local_dh`的顺序就可以轻松纠正这个问题。`42 6d 41 6c`变为`6c 41 6d 42`；`41 64`变为`64 41`。我们的整个字符串变为`6c 41 6d 42 64 41 00`，正确的字符串变为`6e 44 6f 45 69 41 00`，ASCII字符是`nDoEiA`。

恭喜您完成本教程的这一部分。您现在已经拥有了静态逆向工程所需的所有技术！不要忘记通过做练习来巩固你的技能。

## crackme04.c

现在您已经知道了对这些CrackMe进行逆向工程所需的所有工具和技术，我只是要强调每个CrackMe中最重要的部分。解决crackme04可以使用与之前一样的基本过程：在

```
movsx eax, al
add esi, eax
add ecx, 1
```

```
movsxd rax, ecx
movzx eax, byte [rdx + rax]
test al, al
jne 0x72e;[ge]
```

如果al（输入字符串中的一个字节）不为零，那么跳转返回到顶部。否则将ecx与0x10进行比较，如果不相等则失败退出。如果相等则进行另一个检查：如果esi不等于0x6

那么ecx和esi里面是什么？很容易看出ecx是一个计数器。在每次循环迭代中它会递增，并用于索引输入字符串。因此，在循环完成后，它等于字符串中非零字节的数量。

esi仅在循环中的一行被修改：它是字符串中字符数值的总和。它后来与0x632进行了比较（译者注：这里应该是0x6e2而不是0x632，应为作者笔误）。所以我们需要一个

我的方法是简单地做除法然后最后一个字符添加它的余数。1913除以16等于110余数2（译者注：应该是1762而不是1913，又一处笔误），所以我们使用字符110（'n'），

## 附录

### Makefile

使用的Makefile相当简单，但可能有一些难以理解的地方。其中最主要的是在编译后的可执行文件上使用objcopy。我用它来去除FILE符号，否则Radare会利用这个符号在

### 练习

命名为crackme01e.c，crackme02e.c等等的文件是其没有e后缀的对应文件的修改版本。它们用于练习，可以用与本教程各个部分中提到的完全相同的技术来解决。如果

### 媒体报道

2018年1月6日星期六：本教程在Hackaday上被发布，导致我的服务器短暂地宕机。从那里来的朋友们你们好。请看看我的其他教程，如果您想要我创建更多这样的内容，