用Radare2逆向一个自修改的二进制文件

转载

m0_49794426 于 2021-03-24 11:44:53 发布 135 收藏 1
文章标签： vim
原文链接：https://blog.fxti.xyz/2019/02/12/radare2-reverse-self-modified-binary/

序言
写这篇文章花了我三个月，我的TODO列表上有太多的任务以致于这篇文章被放到了最后才开始写。上周我下定决心，到周日我就写完这篇文章。然后我做到了，给你们带来了又一篇Radare2的指南。

今天我们要完成一个有趣的挑战，名叫”packedup”，是ad3l为r2con2017比赛写的。这不是我为r2con比赛写的第一篇writeup，你也可以看看用Radare2逆向一个GameboyROM，确保你不会错过我赢得那次比赛的过程中的收获与体会。

这篇文章是为已经熟悉Radare2的人所写的。如果你不是，我建议你从系列“Radare2之旅”的第一部分(安全客)开始。

闲话少叙，让我们开始分析这个二进制文件。

获取Radare2
安装
Radare2的开发非常迅速，这个项目每天都在更新所以建议你使用最新的github版本，不要使用stable版，因为有时候stable版可能还没有最新的github 版稳定。

```
$ git clone https://github.com/radare/radare2.git
$ cd radare2
$ ./sys/install.sh
```

如果你不想使用github版，或者想要每个平台(Windows,OS X,iOS,等等)相对应的二进制文件，那就点击Radare2下载页面去下载。

升级
正如前面所说的，强烈建议使用从git仓库获取的最新版的r2。这样你只需要用git执行如下指令就行了。

```
$ ./sys/install.sh
```

我经常通过定时任务在早上自动更新Radare2，这样每天用的都是最新版。如果你经常用Radare2，我建议你也这么做。

packedup
你可以从这里下载packedup。我建议你给这个仓库一个星星(★)来获得这个系列的更多更新。

首先运行一下这个文件看看情况。

```
$ ./packedup
Welcome to packedup for r2crackmes □
Flag << MEGABEETS
Try again!
```

packedup执行后要求我们输入flag。它很可能之后进行一些计算来确定我们是否输入了正确的flag。我输入”MEGABEETS”然后输出了错误信息”Try again!”

开始逆向！
开始我们最喜欢的部分，用Radare2打开文件然后弄清楚它怎么检查提交的flag

```
$ r2 ./packedup
— Here be dragons.
[0x004004d0]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn. and sym.func.* functions (aan)
```

分析

我经常起手执行aa(分析全部)或者aas(分析函数、符号等)。这缩写有些误导因为实际上分析任务很繁杂(具体可以查看aa?)，但这对于我测试过的二进制文件这已经够了。因为这次文件小，我直接通过aaa一次搞定。你也能通过-A参数来运行radare2来启动执行aaa(例如r2 -A ./packedup)。

注意：正如我在前一篇博文中提到过的，因为分析过程非常复杂，所以启动执行aaa并非推荐实践。具体可以看看我在这个回答中的详细解释。

获取信息

用Radare2打开文件后，自动定位到程序入口点。但在开始分析代码之前最好先看看文件信息。Radare2可以用i指令显示信息(为了可读行简化过了)

```
[0x004004d0]> i
…
file ./packedup
format elf64
iorw false
mode -r-x
size 0x1878
humansz 6.1K
type EXEC (Executable file)
arch x86
…
bintype elf
bits 64
…
endian little
…
intrp /lib64/ld-linux-x86-64.so.2
lang c
…
machine AMD x86-64 architecture
stripped true
…
```

i指令用来显示已打开文件的信息。该指令使用rabin2这个radare2框架中的信息提取工具。Radare2提供了该文件的大量信息。用i?指令查看更多子命令。

packedup是个64位去符号表ELF文件。有意思，继续分析。

字符串

我最喜欢开始分析的方式之一便是查找可疑的字符串，尤其是引用这些字符串的位置。用iz列出data区段的全部字符串：

```
[0x004004d0]> iz
001 0x000007d0 0x004007d0 46 47 (.rodata) ascii Welcome to packedup for r2crackmes □\nFlag <<
002 0x00000800 0x00400800 11 12 (.rodata) ascii Try again!\n
003 0x0000080d 0x0040080d 26 27 (.rodata) ascii Yep! you got the flag \n
```

不错，可以看到banner字符串以及成功和失败信息。再看看是哪里的代码引用了它们，我们可以使用Radare2内置的循环机制：

```
[0x004004d0]> axt @@ str.*
main 0x400619 [data] mov esi, str.Welcome_to_packedup_for_r2crackmes_:__Flag
main 0x4006b8 [data] mov esi, str.Try_again
main 0x4006de [data] mov esi, str.Yep__you_got_the_flag:
```

这条指令展现了Radare2更多的特性。axt指令用来查找引用这个地址有关的代码或数据引用(可查看ax?)。
@@类似于迭代器符号，用来对于一个列表中的每一个偏移地址重复执行指令(可查看@@?)。然后str.*是对于所有str.开始的标记的通配符。
所以这一整条指令不仅列出了字符串本身还包括引用它们的函数名和相关的代码位置。

定位

之前的字符串都是由main函数所引用的，为了导航到对应位置要使用seek指令s。通过在几乎所有指令后加?就能看到详细的帮助信息。

```
[0x08048370]> s?
|Usage: s # Seek commands
| s Print current address
| s addr Seek to address
| s- Undo seek
| s- n Seek n bytes backward
| s– Seek blocksize bytes backward
| s+ Redo seek
| s+ n Seek n bytes forward
| s++ Seek blocksize bytes forward
| s[j*=] List undo seek history (JSON, =list, *r2)
| s/ DATA Search for next occurrence of ‘DATA’
| s/x 9091 Search for next occurrence of \x90\x91
| s.hexoff Seek honoring a base from core->offset
| sa [[±]a] [asz] Seek asz (or bsize) aligned to addr
| sb Seek aligned to bb start
| sC[?] string Seek to comment matching given string
| sf Seek to next function (f->addr+f->size)
| sf function Seek to address of specified function
| sg/sG Seek begin (sg) or end (sG) of section or file
| sl[?] [±]line Seek to line
| sn/sp Seek next/prev scr.nkey
| so [N] Seek to N next opcode(s)
| sr pc Seek to register
```

所以基本上，seek指令接受地址或者数学表达式作为参数。定位到main函数很简单，直接执行s main即可。

```
[0x004004d0]> s main
[0x0040060c]>
```

反汇编

现在位于main函数需要打印汇编代码看看。Radare2提供了多种方式反汇编一个函数。可以用pdf(显示函数反汇编)或者交互性更强的可视化模式(v)和图形模式(VV)，选哪个你喜欢就好。我惯用可视化模式因为信息更加丰富，交互性也好。我在上一篇详细讲解过，所以这里只是使用。

让我们看看main函数的开始部分：

```
[0x0040060c]> pdf
 ┌ (fcn) main 244
 │ main ();
 │ ; var int local_10h @ rbp-0x10
 │ ; var int local_ch @ rbp-0xc
 │ ; var int local_8h @ rbp-0x8
 │ ; JMP XREF from 0x00400605 (entry2.init)
 │ ; DATA XREF from 0x004004ed (entry0)
 │ 0x0040060c 55 push rbp
 │ 0x0040060d 4889e5 mov rbp, rsp
 │ 0x00400610 4883ec10 sub rsp, 0x10
 │ ; DATA XREF from 0x00400653 (main)
 │ ; DATA XREF from 0x004005e7 (entry2.init)
 │ 0x00400614 ba30000000 mov edx, 0x30
 │ 0x00400619 bed0074000 mov esi, str.Welcome_to_packedup_for_r2crackmes_:___Flag ; 0x4007d0 ; “Welcome to packedup for r2crackmes □\nFlag << “
 │ 0x0040061e bf01000000 mov edi, 1
 │ 0x00400623 b800000000 mov eax, 0
 │ 0x00400628 e853feffff call sym.imp.write ; ssize_t write(int fd, void *ptr, size_t nbytes)
 │ 0x0040062d ba2c000000 mov edx, 0x2c ; ‘,’ ; 44
 │ 0x00400632 be80106000 mov esi, 0x601080
 │ 0x00400637 bf00000000 mov edi, 0
 │ 0x0040063c b800000000 mov eax, 0
 │ 0x00400641 e84afeffff call sym.imp.read ; ssize_t read(int fildes, void *buf, size_t nbyte)
 │ 0x00400646 4898 cdqe
 │ 0x00400648 488945f8 mov qword [local_8h], rax
 │ 0x0040064c c745f0000000. mov dword [local_10h], 0
 │ 0x00400653 b814064000 mov eax, 0x400614
 │ 0x00400658 bbf6064000 mov ebx, 0x4006f6
 │ 0x0040065d 29c3 sub ebx, eax
 │ 0x0040065f 31c9 xor ecx, ecx
```

在函数初始化之后，能看到参数传递到write()打印出banner信息然后另一个参数传递给read()从stdin读取输入。read()从stdin读取0x2c(十进制:44)字节存储到位于0x601080的缓冲区。假设flag很可能长为44个字符，重命名保存输入的地址来方便自己识别。

```
[0x0040060c]> f loc.our_input = 0x601080
```

f用来为一个特定地址创建flag(标记)。更多信息可查看f?并阅读这一章。

然后程序保存读入字节数到一个本地变量。在0x400653地址0x400614存入eax，地址0x4006f6存入ebx。第一个地址是main函数初始化部分之后的地址，第二个是main函数返回之前的地址。之后ebx减去eax所以ebx中存储着两个地址的差值。

下一部分，程序使用在一个循环中使用自身的代码计算出了一个初始值。

…

```
│ 0x00400653 b814064000 mov eax, 0x400614
│ 0x00400658 bbf6064000 mov ebx, 0x4006f6
│ 0x0040065d 29c3 sub ebx, eax
│ 0x0040065f 31c9 xor ecx, ecx
│ ; JMP XREF from 0x0040066b (main)
│ ┌─> 0x00400661 670208 add cl, byte [eax]
│ │ 0x00400664 c1c904 ror ecx, 4
│ │ 0x00400667 ffc0 inc eax
│ │ 0x00400669 ffcb dec ebx
│ └─< 0x0040066b 75f4 jne 0x400661
│ 0x0040066d 89ca mov edx, ecx
```

…
在下个循环中能看到程序在验证输入。首先看看最相关的部分

```
│ 0x0040066d 89ca mov edx, ecx
│ 0x0040066f 89d0 mov eax, edx
│ 0x00400671 8945f0 mov dword [local_10h], eax
│ 0x00400674 c745f42c0000. mov dword [local_ch], 0x2c ; ',' ; 44
│ ; JMP XREF from 0x004006f4 (main)
│ 0x0040067b 8b45f0 mov eax, dword [local_10h]
│ 0x0040067e 89c0 mov eax, eax
│ 0x00400680 d1c0 rol eax, 1
│ 0x00400682 8945f0 mov dword [local_10h], eax
│ 0x00400685 8b45f0 mov eax, dword [local_10h]
│ 0x00400688 0fb6d0 movzx edx, al
│ 0x0040068b 8b45f4 mov eax, dword [local_ch]
│ 0x0040068e 83e801 sub eax, 1
│ 0x00400691 4898 cdqe
│ 0x00400693 0fb688801060. movzx ecx, byte [rax + 0x601080] ; [0x601080:1]=0
│ 0x0040069a 8b45f4 mov eax, dword [local_ch]
│ 0x0040069d 83e801 sub eax, 1
│ 0x004006a0 4898 cdqe
│ 0x004006a2 0fb680a00740. movzx eax, byte [rax + 0x4007a0] ; [0x4007a0:1]=15
│ 0x004006a9 31c8 xor eax, ecx
│ 0x004006ab 0fb6c0 movzx eax, al
│ 0x004006ae 39c2 cmp edx, eax
│ ┌─< 0x004006b0 741c je 0x4006ce
```

在之前循环中计算的初始值被移入edx(0x40066d)，之后又先进入eax再进入[local_10h]。再进入验证循环，初始值从[local_10h]到eax，然后eax被左移1字节(0x400680)然后再次被移到[local_10h]。在0x400688低字节al中的值被移到edx。在0x400693程序从输入取1字节到ecx。在0x4006a2，程序从[rax + 0x4007a0]取1字节到eax。看看0x4007a0有什么：

```
[0x0040060c]> px 44 @ 0x4007a0
– offset – 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x004007a0 0fc9 a886 ace0 1893 8aaf 91a2 e464 7a5a …dzZ
0x004007b0 088b a89a b4d1 1f84 b4d1 7152 1c8a 80d2 …qR…
0x004007c0 1495 8297 80d2 1f90 8f91 93a6 …
```

不错，这是预定义的44字节数组，定然用在了比较过程中。给它取个名字叫loc.predefined_array：

[0x0040060c]> f loc.predefined_array = 0x4007a0
再继续之前提醒一下，edx保存着移位过的初始值，eax保存着loc.predefined_array[0x2c-1]的1字节内容,ecx保存着输入字符串的最后一个字符。

在0x4006a9，ecx和eax位异或结果存储在eax。然后al拓展到eax并且和edx比较(移位过的初始值)。可以写出如下Python伪代码：

key = 0x??? # Our initial_value
our_input = “A” * 44
predefined_array = [0x0fc9, … ,0x93a6]

if (ord(our_input[-1]) ^ predefined_array[-1]) == (key & 0xff):
return True
如果结果相等，比如ecx ^ eax == edx，循环以输入的倒数第二个字母和loc.predefined_array[0x2c-2]开始并且和rol(key)比较。key实际上是左移的initial_value，正因为这个值每次左移都会改变所以被称为key。

第一次求解
首先我们定义一个函数完成左移：

Rotate left lambda

rol = lambda val, r_bits, max_bits:
(val << r_bits%max_bits) & (2**max_bits-1) |
((val & (2max_bits-1)) >> (max_bits-(r_bits%max_bits)))
然后创建出预定义的数组，用Radare2完成：

[0x0040060c]> pcp 44 @ 0x4007a0
import struct
buf = struct.pack (“44B”, *[
0x0f,0xc9,0xa8,0x86,0xac,0xe0,0x18,0x93,0x8a,0xaf,0x91,
0xa2,0xe4,0x64,0x7a,0x5a,0x08,0x8b,0xa8,0x9a,0xb4,0xd1,
0x1f,0x84,0xb4,0xd1,0x71,0x52,0x1c,0x8a,0x80,0xd2,0x14,
0x95,0x82,0x97,0x80,0xd2,0x1f,0x90,0x8f,0x91,0x93,0xa6])
pc子命令用来按字节通过C, Python, Javascript等语言的格式输出文件内数据(详细可看pc?)。

把这个加入脚本并定义变量

Byte array from 0x004007A0, modified from the generated results of `pcp 44 @ 0x4007a0`

arr =[0x0F, 0xC9, 0xA8, 0x86, 0xAC, 0xE0, 0x18, 0x93, 0x8A, 0xAF, 0x91, 0xA2, 0xE4, 0x64, 0x7A, 0x5A, 0x08, 0x8B, 0xA8, 0x9A, 0xB4, 0xD1, 0x1F, 0x84, 0xB4, 0xD1, 0x71, 0x52, 0x1C, 0x8A, 0x80, 0xD2, 0x14, 0x95, 0x82, 0x97, 0x80, 0xD2, 0x1F, 0x90, 0x8F, 0x91, 0x93, 0xA6]

Initial key value

key = 0xdc77df87

flag = []
最后实现逻辑然后组合在一起：

Rotate left lambda

rol = lambda val, r_bits, max_bits:
(val << r_bits%max_bits) & (2**max_bits-1) |
((val & (2max_bits-1)) >> (max_bits-(r_bits%max_bits)))

Byte array from 0x004007A0, modified from the generated results of `pcp 44 @ 0x4007a0`

arr =[0x0F, 0xC9, 0xA8, 0x86, 0xAC, 0xE0, 0x18, 0x93, 0x8A, 0xAF, 0x91, 0xA2, 0xE4, 0x64, 0x7A, 0x5A, 0x08, 0x8B, 0xA8, 0x9A, 0xB4, 0xD1, 0x1F, 0x84, 0xB4, 0xD1, 0x71, 0x52, 0x1C, 0x8A, 0x80, 0xD2, 0x14, 0x95, 0x82, 0x97, 0x80, 0xD2, 0x1F, 0x90, 0x8F, 0x91, 0x93, 0xA6]

Initial key value

key = 0x???

flag = []

Iterate the array backwards

for b in reversed(arr):
# rol the key
key = rol(key,1,32)
# xor the 8 lower bits of the key with current byte in array
char = chr(b ^ (key & 0xff))
# Add char to final flag
flag.insert(0, char)

print '[+] Flag: ', ''.join(flag)
不错，现在只需要获取初始值就行了。应该在初始值生成循环后下断点然后查看即可。Radare2调试模式启动r2 -d packedup或者已经在radare2里就用ood指令。

[0x0040060c]> ood
Process with PID 1236 started…
File dbg:///home/beet/Desktop/Security/r2con/crackmes/packedup reopened in read-write mode
= attach 1236 1236
[0x7fb45818cf30]>
ood指令将在调试模式重新打开当前文件。它是oo指令(负责处理已打开文件)的子指令。

可以看到当前定位被更改成了0x7fb45818cf30，这是动态加载器(ld.so)内存中的地址。通过dm.指令就能方便的知道:

[0x7fb45818cf30]> dm.
/usr/lib/ld-2.26.so
dm. is used to show the
dm.用来显示当前地址映射到的区块名。用dm能列出当前进程的内存映射信息，用dm?能看到更多子指令。

已知循环尾端在0x0040066d，计算结果在ecx，正要移到edx。设定断点来查看ecx内容:

[0x7fb45818cf30]> db 0x40066d
[0x7fb45818cf30]> dc
Welcome to packedup for r2crackmes □
Flag << Blah_blah_blah
hit breakpoint at: 40066d

[0x0040066d]> dr ecx
0xd477d83e
db负责设定调试断点，db?可看到其子指令。
dc负责调试时继续执行。
dr负责显示调试时寄存器的值，db?可看到其子指令。
d?可以看和调试相关的所有指令

那么把初始值填入脚本key = 0xd477d83e并执行:

beet:~$ python answer.py
[+] Flag: Hj.xlgYj{u]J*l=ok\S6?TJbgh
什么？这flag看起来不可能是对的。分析过程肯定哪里出了问题！

哦对了！我怎么没想到！还记得之前说过的吗？

下一部分，程序使用在一个循环中使用自身的代码计算出了一个初始值。

把这个给忘了。使用软断点db 0x40066d事实上在代码中加入了CC(INT3)，这改变了原始代码导致了初始值计算出错。INT3指令生成一个特殊的单字节汇编指令CC用来调用调试器完成(异常)处理。

第二次求解
所以这次使用硬断点，禁用之前设定的软断点并设置硬断点:

[0x40066d]> db- 0x40066d
[0x40066d]> ood
Process with PID 1317 started…
File dbg:///home/beet/Desktop/Security/r2con/crackmes/packedup reopened in read-write mode
= attach 1317 1317

[0x7fcb28f77f30]> drx 1 0x0040066d 1 x
[0x7fcb28f77f30]> dc
Welcome to packedup for r2crackmes
Flag << Blah_Megabeets_Blahaaa

[0x0040066d]> dr ecx
0xdc77df87
重新填入初始值并再次尝试:

beet:~$ python answer.py
[+] Flag: ['\xc8', '*', '\xd9', '>', 'p', '\x0e', '\xef', 'h', '\xf7', '\x91', '\x8e', '\xad', 'c', '\xa7', '\x9b
别这样啊…又错了。我还遗漏了什么？

既然还在调试模式，那么从验证循环的开始0x0040067b单步执行检查一遍。

汇编:0x00400680 d1c8 ror eax, 1
解释:

然后在0x00400680，eax被左移了1字节

慢着！之前分析是左移1字节(rol eax, 1)，但现在是右移1字节(ror eax, 1)？！难怪得到了错误的flag，但是这是什么时候改变的？

为了弄明白这个问题，在0x00400680上设定硬件断点检测写入访问来看看是什么时候改变的。

```
[0x40066d]> doo
Process with PID 1611 started…
File dbg:///home/beet/Desktop/Security/r2con/crackmes/packedup reopened in read-write mode
= attach 1611 1611

[0x7fc43ae24f30]> drx 1 0x00400680 1 w
[0x7fc43ae24f30]> dc
[0x00400600]>
```

调试停止在0x400600，打印汇编看看:

```
[0x00400600]> pd 1
0x00400600 67c6406dc8 mov byte [eax + 0x6d], 0xc8
```

就是这个修改了该地址上的汇编指令，rol eax, 1对应的应该是dlc0，这里改成了dlc8！我们可以使用radare2反汇编16进制字符串看看:

```
[0x00400600]> pad d1c0
rol eax, 1
[0x00400600]> # changed to
[0x00400600]> pad d1c8
ror eax, 1
```

打印整个函数看看:

```
[0x00400600]> pdf
 ┌ (fcn) entry2.init 77
 │ entry2.init ();
 │ 0x004005bd 90 nop
 │ 0x004005be 90 nop
 │ 0x004005bf 90 nop
 │ 0x004005c0 90 nop
 │ 0x004005c1 b87d000000 mov eax, 0x7d ; '}' ; 125
 │ 0x004005c6 bbff0f0000 mov ebx, 0xfff
 │ 0x004005cb f7db neg ebx
 │ 0x004005cd ffcb dec ebx
 │ 0x004005cf 81e314064000 and ebx, 0x400614
 │ 0x004005d5 b9f6064000 mov ecx, 0x4006f6
 │ 0x004005da 81e914064000 sub ecx, 0x400614
 │ 0x004005e0 ba07000000 mov edx, 7
 │ 0x004005e5 cd80 int 0x80
 │ 0x004005e7 b814064000 mov eax, 0x400614
 │ 0x004005ec 67c64050c1 mov byte [eax + 0x50], 0xc1 ; [0xc1:1]=255 ; 193
 │ 0x004005f1 67c64051c1 mov byte [eax + 0x51], 0xc1 ; [0xc1:1]=255 ; 193
 │ 0x004005f6 67c6405203 mov byte [eax + 0x52], 3
 │ 0x004005fb 67c6406cd1 mov byte [eax + 0x6c], 0xd1 ; [0xd1:1]=255 ; 209
;– rip:
 │ 0x00400600 67c6406dc8 mov byte [eax + 0x6d], 0xc8 ; [0xc8:1]=255 ; 200
 └ ┌─< 0x00400605 e902000000 jmp main
```

可以看到这个函数被Radare2命名为entry2.init。能看到在0x4005c1上0x7d移入eax然后在0x4005e5上syscall(int 80)指令被执行。这调用了mprotect函数修改.text区段权限使其可写，之后再修改main函数中的rol变成ror。

.init_array

entry2.init是初始化函数，在main函数之前执行，这就是为什么会遗漏它。

在执行到main函数之前，运行时链接器执行该程序初始化段(.preinit_array和.init_array)中保存的地址所指定的任意函数。这些函数通常是构造函数，同理.fini_array段中保存的是程序正常退出前需要执行的析构函数。

Radare2同样是识别出了.init_array段:

[0x00400600]> f~init_array
0x00600e08 16 section…init_array
0x00600e18 0 section_end…init_array
[0x00400600]> pxW 24 @ section…init_array
0x00600e08 0x00400590 entry1.init
0x00600e0c 0x00000000 section.
0x00600e10 0x004005bd entry2.init
0x00600e14 0x00000000 section.
0x00600e18 0x00400570 entry3.fini
0x00600e1c 0x00000000 section.
f用来列出radare创建的标记
~是radare内建的grep
pxW按16进制打印双字(32位)

能看到其中有两个函数，包括entry2.init。

顺带一提，在gcc中定义初始化函数可以使用这个类似的模板:

void **attribute** ((constructor)) some_func() {

```
// code goes here
```

}
第三次求解
现在把脚本中的rol变成ror:

Rotate right lambda

ror = lambda val, r_bits, max_bits:
((val & (2**max_bits-1)) >> r_bits%max_bits) |
(val << (max_bits-(r_bits%max_bits)) & (2max_bits-1))

Byte array from 0x004007A0, modified from the generated results of `pcp 44 @ 0x4007a0`

arr = [
0x0F, 0xC9, 0xA8, 0x86, 0xAC, 0xE0, 0x18, 0x93, 0x8A, 0xAF, 0x91, 0xA2,
0xE4, 0x64, 0x7A, 0x5A, 0x08, 0x8B, 0xA8, 0x9A, 0xB4, 0xD1, 0x1F, 0x84,
0xB4, 0xD1, 0x71, 0x52, 0x1C, 0x8A, 0x80, 0xD2, 0x14, 0x95, 0x82, 0x97,
0x80, 0xD2, 0x1F, 0x90, 0x8F, 0x91, 0x93, 0xA6
]

Initial key value

key = 0xdc77df87

flag = []

Iterate the array backwards

```
for b in reversed(arr):
# ror the key
key = ror(key, 1, 32)
# xor the 8 lower bits of the key with current byte in array
char = chr(b ^ (key & 0xff))
# Add char to final flag
flag.insert(0, char)

print '[+] Flag: ', ''.join(flag)
```

第三次执行！

```
beet:~$ python answer.py
[+] Flag: r2_is_for_packedup_things_like_linux_malware
```

求解完毕！

尾声

如果你想更多了解Radare2建议看看Radare2之旅的第一部分和第二部分