Shellcoding Lab 64 BIT (0x0f05)

0x7E0 @ berlinsides edition
by dash

HACK4

Wait! What?

- no it is still *not* shellscripting
- name comes from gaining a shell
- instructions are passed to cpu
- no extra compiling or linking needed

→ if injected into a process

Typical Usage

Remote:

- you want to gain code execution remotely

Local:

- privilege escalation

Userland:

- Pretty playing, good training

Is this fun?!

- Enjoying assembly!

→ great to understand whats going on

→ coming from C its easy for you

- Exploit some or everything!

→ no chance without understanding a piece of assembly

Is this fun?!

- Own a careless internet user!

→ I have this awesome ssh remote r00t here

- Be on a uberc00l hacker con and blather about!

→ Hi Aluc!

Basics

- 64Bit ***is*** different from 32Bit
- 64Bit / 8 byte / 16 nibble
- different calling convention
- different usage of registers
- different syscall numbers

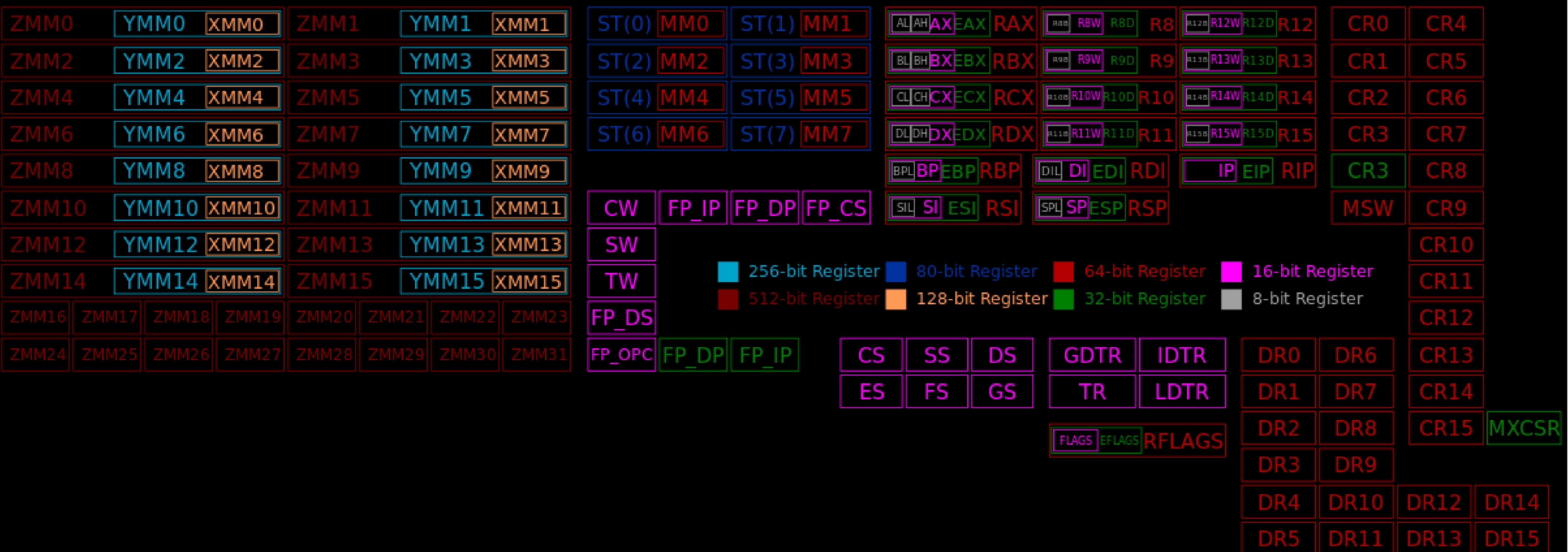
Registers

- So, we have 64Bit Registers now!
- 16 Registers there are, young Padawan.

→ instead of having 8 on IA-32

- new Registers (almost) new names
- some look quite familiar

Registers
(from Wikipedia)



32BIT → 64BIT

EAX → RAX – Accumulator

ECX → RCX – Count Register

EDX → RDX – Data Register

EBX → RBX – Base Register

ESI → RSI – Stream Source

EDI → RDI – Stream Destination

ESP → RSP – Stack Pointer

EBP → RBP – Base Pointer

Registers

RAX – Accumulator

RCX – Count Register

RDX – Data Register

RBX - Base Register

RSI – Stream Source

RDI – Stream Dest.

R8 – R15 (new)

RSP – Stack Pointer

→ Points to next instruction

RBP – Base Pointer

→ start of current stack frame

A lot of words

- b is byte (yes, 1 **whole** byte ;))
- w is word (2 byte)
- d double word (4 byte)
- q quad word (8 byte)

→ *note* there is no rXq for full addressing, its rX (e.g. r10)

→ also former general purpose registers go by their former name (e.g. rax → eax → ax → ah → al)

→ *note* for GDB it's giant word (e.g. x/20g $rsp)

Registers – Syscall Arguments

Register	Purpose	Other
RAX	Syscall	Return Value!
RDI	1. Argument	
RSI	2. Argument	
RDX	3. Argument	
R10	4. Argument	
R8	5. Argument	
R9	6. Argument	

Registers

- Different types of addressing!
- Register ↔ Argument!
- RAX also gets a return value (if not void)
- Legacy Registers just have an R now

→ simple, no?

Instructions

xor rax, rax

→ null out register / xor register with value

mov rax, 60

→ copy decimal 60 into rax register

xchg rax,rbx

→ exchange register with register

Instructions

inc rax

→ increase (+1) in rax

dec rax

→ decrease (-1) in rax

nop

→ mostly known from exploits, “no operation”

Instructions

add rax,1

→ plus one to rax

sub rax,1

→ subtract one from rax

adc rax,1

→ add one to rax, but also check carry flag

Instructions

jmp

→ go to a subfunction (short jmp 1byte / near jmp 2byte)

call h3ll

→ calls a subfunction

ret

→ this re-establishes stack pointer, (mov rsp, rbp)

Instructions

push 0x41424344

→ push 4 byte on the stack

pop rsi

→ get the data from stack and fill it into rsi

→ stack consumption decreases to higher address

syscall

→ it is NOT anymore int 0x80, we just use 'syscall'

Now

→ 0x0f05 is the bytecode (not anymore 0xcd80)

Cold Water plz

```
BITS 64
global _start
_start:
mov r10b,10
mov r10,10
mov r9,9
mov r11w,8000
mov r12d,0x41424344
mov r13,0x4142434445464748

xor rax, rax
mov al, 60
syscall
```

Compile it:

1. nasm -f elf64 -o test.o test.asm
2. nasm -f elf64 -o test2.o test.asm -O0

Compare both in objdump:

1. objdump -d test.o -M intel
2. objdump -d test2.o -M intel

Whats going on here?

Cold Water plz

With Optimization	W/O Optimization

```
0:   41 b2 0a                 mov    r10b,0xa
3:   41 ba 0a 00 00 00        mov    r10d,0xa
9:   41 ba 0a 00 00 00        mov    r10d,0xa
f:   41 b9 09 00 00 00        mov    r9d,0x9
15:  66 41 bb 40 1f           mov    r11w,0x1f40
1a:  41 bc 44 43 42 41        mov    r12d,0x41424344
20:  49 bd 48 47 46 45 44     movabs
r13,0x4142434445464748
27:  43 42 41
2a:  48 31 c0                 xor    rax,rax
2d:  b0 3c                    mov    al,0x3c
2f:  0f 05                    syscall
```

```
0:   41 b2 0a                 mov    r10b,0xa
3:   49 ba 0a 00 00 00 00 00  movabs r10,0xa
a:   00 00 00
d:   49 ba 0a 00 00 00 00 00  movabs r10,0xa
14:  00 00 00
17:  49 b9 09 00 00 00 00 00  movabs r9,0x9
1e:  00 00 00
21:  66 41 bb 40 1f           mov    r11w,0x1f40
26:  41 bc 44 43 42 41        mov    r12d,0x41424344
2c:  49 bd 48 47 46 45 44     movabs
r13,0x4142434445464748
33:  43 42 41
36:  48 31 c0                 xor    rax,rax
39:  b0 3c                    mov    al,0x3c
3b:  0f 05                    syscall
```

Recap

- if not stated otherwise nasm will optimize the code

- use -O0 to disable optimization

- if you addressed rax, but the code uses eax, check for enabled optimization

→ check for different results with and without optimization

Gdb - short

- gnu debugger
- available on all linux platforms and most unix*s
- not as nice as immunity debugger, but it does its job
- gdb ./<name> -q
- quitemode, we dont need the rest
- normal mode of gdb
- most commands have abbreviations

Gdb - short

- break / b

→ set breakpoints, break _start / break main

- run / r

→ run forrest run!

- info registers / i r

→ show general purpose registers and segments

- disassembly / disas

→ current position in code

GDB

```
$ gdb ./xchg -q
Reading symbols from ./xchg...(no debugging symbols found)...done.
(gdb) break _start
Breakpoint 1 at 0x400080
(gdb) run
Starting program: /home/user//Shellcode-Lab/64BIT/exchange_registers/xchg
Breakpoint 1, 0x0000000000400080 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x0000000000400080 <+0>:     xor    rax,rax
   0x0000000000400083 <+3>:     xor    rbx,rbx
   0x0000000000400086 <+6>:     movabs rax,0x29a
   0x0000000000400090 <+16>:    movabs rbx,0x539
   0x000000000040009a <+26>:    movabs r10,0xbeefbeefbeefbeef
   0x00000000004000a4 <+36>:    xchg   r10,rax
   0x00000000004000a6 <+38>:    xchg   r9,r10
   0x00000000004000a9 <+41>:    xchg   rbx,rax
   0x00000000004000ab <+43>:    xchg   rsp,rdi
End of assembler dump.
```

```
(gdb) info registers
rax      0x0    0
rbx      0x0    0
rcx      0x0    0
rdx      0x0    0
rsi      0x0    0
rdi      0x0    0
rbp      0x0    0x0
rsp      0x7fffffffea40   0x7fffffffea40
r8       0x0    0
r9       0x0    0
r10      0x0    0
r11      0x0    0
r12      0x0    0
r13      0x0    0
r14      0x0    0
r15      0x0    0
rip      0x400080 0x400080 <_start>
eflags   0x202   [ IF ]
cs       0x33    51
ss       0x2b    43
ds       0x0    0
es       0x0    0
fs       0x0    0
gs       0x0    0
```

Gdb - short

- step / s

→ until exit from function

- stepi / si

→ step instructions (we want that!)

- i r rax rbx r10

→ info registers only accumulator, base and r10

→ press enter again

→ last command will be repeated

Gdb Intro

```
Breakpoint 1, 0x0000000000400080 in _start ()
(gdb) si
0x0000000000400083 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:    xor    rax,rax
=> 0x0000000000400083 <+3>:    xor    rbx,rbx
   0x0000000000400086 <+6>:    movabs rax,0x29a
   0x0000000000400090 <+16>:   movabs rbx,0x539
   0x000000000040009a <+26>:   movabs r10,0xbeefbeefbeefbeef
   0x00000000004000a4 <+36>:   xchg   r10,rax
   0x00000000004000a6 <+38>:   xchg   r9,r10
   0x00000000004000a9 <+41>:   xchg   rbx,rax
   0x00000000004000ab <+43>:   xchg   rsp,rdi
End of assembler dump.
(gdb) si
0x0000000000400086 in _start ()
(gdb) si
0x0000000000400090 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x0000000000400080 <+0>:    xor    rax,rax
   0x0000000000400083 <+3>:    xor    rbx,rbx
   0x0000000000400086 <+6>:    movabs rax,0x29a
=> 0x0000000000400090 <+16>:   movabs rbx,0x539
   0x000000000040009a <+26>:   movabs r10,0xbeefbeefbeefbeef
   0x00000000004000a4 <+36>:   xchg   r10,rax
   0x00000000004000a6 <+38>:   xchg   r9,r10
   0x00000000004000a9 <+41>:   xchg   rbx,rax
   0x00000000004000ab <+43>:   xchg   rsp,rdi
End of assembler dump.
```

```
(gdb) info registers rax rbx rcx
rax            0x29a    666
rbx            0x0      0
rcx            0x0      0
(gdb) si
0x000000000040009a in _start ()
(gdb)
0x00000000004000a4 in _start ()
(gdb)
0x00000000004000a6 in _start ()
(gdb) info registers rax rbx rcx
rax            0xbeefbeefbeefbeef
-4688318750159552785
rbx            0x539    1337
rcx            0x0      0
```

GDB Intro

```
BITS 64
global _start

_start:

xor rax, rax
xor rbx, rbx

mov rax, 0x29A ; http://web.textfiles.com/ezines/29A/
mov rbx, 0x539
mov r10, 0xBEEFBEEFBEEFBEEF
xchg rax, r10
xchg r10, r9
xchg rbx, rax
xchg rdi,rsp
```

Compile it:

$ nasm -f elf64 -o xchg.o xchg.asm -O0

$ ld -o xchg xchg.o

Debug it with gdb.

Byte Placement

- please check the both example codes in gdb

→ byte_placement_rax.asm

→ byte_placement_r10.asm

- what is the difference?

Syscall Examples

32BIT		64Bit	
exit	1	exit	60
read	3	read	0
write	4	write	1
open	5	open	2
close	6	close	3
execve	11	execve	59
chdir	12	chdir	80
chmod	15	chmod	90
setuid	23	setuid	105
kill	37	kill	62
reboot	88	reboot	169
socket	102	socket	41
connect	102	connect	42
accept	102	accept	43
bind	102	bind	49
listen	102	listen	50

Syscall

- What is a syscall?
- *nix using Syscalls!
- man 2 syscall
- Quite some differences in number 32/64bit

  /usr/include/asm/unistd_32.h

  /usr/include/asm/unistd_64.h

Registers – Syscall Arguments

Register	Purpose	Other
RAX	Syscall	Return Value!
RDI	1. Argument	
RSI	2. Argument	
RDX	3. Argument	
R10	4. Argument	
R8	5. Argument	
R9	6. Argument	

Syscall: exit

- man 2 exit
- void exit (int status)
- look up the syscall in unistd_64
- 60 or 3Ch
- we have one argument and no return code

Convert decimal to hex

- python to rescue

```
python -c 'print hex(60)'
0x3c
```

- commandline

```
$ bc
obase=16
60
3C
```

- a million ways to do that (you could also do that in javascript ;))

Syscall: exit

```
Bits 64
global _start

_start:

mov     rax,0x3C
mov     rdi,4
syscall
```

```
nasm -f elf64 exit.asm -o exit.o
ld -o exit exit.o
```

```
$ ./exit ; echo $?
4
```

Syscall: exit (nasm optimized)

```
Bits 64
global _start
; label _start
_start:

mov     rax,0x3C ; mov 60 to RAX
mov     rdi,4       ; mov 4 into RDI
syscall             ; execute the syscall
```

```
nasm -f elf64 exit.asm -o exit.o
ld -o exit exit.o
```

```
$ ./exit ; echo $?
4
```

```
0000000000400080 <_start>:
400080:    b8 3c 00 00 00    mov    eax,0x3c
400085:    bf 04 00 00 00    mov    edi,0x4
40008a:    0f 05             syscall
```

Syscall: exit
(nasm un-optimized -O0)

```
Bits 64
global _start
; label _start
_start:

mov     rax,0x3C ; mov 60 to RAX
mov     rdi,4       ; mov 4 into RDI
syscall             ; execute the syscall
```

```
nasm -f elf64 exit.asm -o exit.o -O0
ld -o exit exit.o
```

```
$ ./exit ; echo $?
4
```

```
0000000000400080 <_start>:
400080:   48 31 c0            xor   rax,rax
400083:   48 31 d2            xor   rdx,rdx
400086:   b8 3c 00 00 00      mov   eax,0x3c
40008b:   ba 04 00 00 00      mov   edx,0x4
400090:   0f 05               syscall
```

Exploit Skeleton

```
#include <stdio.h>
#include <string.h>

unsigned char code[] =”shellcode wants to be placed here!”;
main()
{
   printf("Shellcode Len: %d\n", (int)strlen(code));
   int (*ret)() = (int(*)())code;
   ret();
}
```

Syscall: exit

- Linux Command Chain (Command Line Fu)

$ objdump -d ./exit|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'

-Shellnoob Tool

$ shellnoob.py --from-obj exit --to-c exit.c

char shellcode[] = "\xb8\x3c**\x00\x00\x00**\xbf\x04**\x00\x00\x00**\x0f\x05";

- Place the shellcode and compile the skeleton

$ gcc -z execstack skeleton.c -o exit_shell

- Execute it

$ ./exit_shell

shellcode len: 2

Why god, whyyy?

- Why is the shellcode not working?
- For some reason the length is too short…
- Reasons:
* compiled it without -z execstack
* null bytes in the code

Nullbytes

- The shellcode won't work this way!
- First we need to get rid of all nullbytes!
- Use only the parts of a register which are needed!
- Try to find alternative ways to use 0 without generating a null byte!

Nullbytes

—------- Write a shellcode without nullbytes! -------

Syscall: exit
(non-optimized by nasm)

```
Bits 64
global _start

_start:

xor     rax,rax
xor     rdx,rdx
mov     al,0x3C
mov     dil,4
syscall
```

```
nasm -f elf64 exit.asm -o exit.o -O0
ld -o exit exit.o
```

```
$ ./exit ; echo $?
4
```

```
0000000000400080 <_start>:
400080:    48 31 c0        xor    rax,rax
400083:    48 31 d2        xor    rdx,rdx
400086:    b0 3c           mov    al,0x3c
400088:    40 b7 04        mov    dil,0x4
40008b:    0f 05           syscall
```

Exit Shellcode in Skeleton

//btw. if variable **shellcode** is **const**, its **placed** in a **different segment** and -z **execstack** is **not needed**

```
gcc skeleton.c -o exit -z execstack
./exit_shell ;echo $?
shellcode len: 13
4

/* skeleton for shellcode testing
   dash@hack4.org
*/

#include <stdio.h>
#include <string.h>

unsigned char code[]="\x48\x31\xc0\x48\x31\xd2\xb0\x3c\x40\xb7\x04\x0f\x05";

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

8BIT Registers
(oh there they are)

- You remember Wikipedia saying there is not 8Bit addressing?

- Well, lets check that again.

8Bit Registers

```
BITS 64

global _start
_start:

mov spl, 1
mov bpl, 2
mov sil, 3
mov dil, 4
```

```
nasm -f elf64 8bit.asm -o 8bit.o -O0
ld -o 8bit 8bit.o
```

```
400080:    40 b4 01    mov    spl,0x1
400083:    40 b5 02    mov    bpl,0x2
400086:    40 b6 03    mov    sil,0x3
400089:    40 b7 04    mov    dil,0x4
```

8BIT Registers
(oh they **are** there)

- So, if you want to address 1byte only – go with that.

Lessons learned

- how to address registers
- use objdump to check your shellcode
- workaround if addressing registers gets nasty
- avoid nullbytes
- keep in mind execstack / noexecstack
- or set char shellcode to constant

Syscall: kill

- man 2 kill (what a cmdline)
- int kill(pid_t pid, int sig);
- pid – process id
- sig – signal

Syscall: kill

```
BITS 64
global _start
_start:
xor rax, rax
xor rdi, rdi
xor rsi, rsi
; fill arguments for syscall kill
mov dil, XXXX ; first argument
mov sil, XXXX ; second argument
mov al, XXXX ; syscall nr
syscall
```

```
nasm -f elf64 kill.asm -o kill.o -O0
ld -o kill kill.o
```

```
$ <process>
Killed
```

Syscall: kill (nasm un-optimized)

```
0000000000400080 <_start>:
400080:    48 31 c0    xor    rax,rax
400083:    48 31 ff    xor    rdi,rdi
400086:    48 31 f6    xor    rsi,rsi
400089:    40 b7 01    mov    dil,0x1
40008c:    40 b6 09    mov    sil,0x9
40008f:    b0 3e       mov    al,0x3e
400091:    0f 05       syscall
```

```
nasm -f elf64 kill_noexit.asm -o kill_noexit.o -O0

ld -o kill_noexit kill_noexit.o
```

```
$ <process>
Killed
```

Syscall: kill

```
$ ./kill_noexit
Segmentation fault (core dumped)
$ strace ./kill_noexit
execve("./kill_noexit", ["./kill_noexit"], [/* 29 vars */]) = 0
kill(1, SIGKILL)                        = -1 EPERM (Operation not permitted)
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x9} ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
```

Syscall: kill

- What happens if you don't have a exit call
- Not only killing a process also:

→ restart, read config, stop or continue

- Killer Shellcode? Kill all processes on the box

→ Don't do that :)

Push

- Push values on the stack
- Yes, you get them into a register with pop
- byte/word/dword/(giant?)

```
[...]
push 0x41
push 0x4142
push 0x41424344
push 0x4142434445464748
[/..]
```

```
nasm -f elf64 push.asm -o push.o; ld -o push push.o
push.asm:14: warning: signed dword immediate exceeds bounds
```

Push

```
BITS 64
global _start
_start:
push 0x41
push 0x4142
push 0x41424344
; lets comment that out
; push 0x4142434445464748 ← try to compile it with
------------------------------------------------ null byte free version:
BITS 64
global _start
_start:
push byte 0x41
push word 0x4142
push dword 0x41424344
```

```
nasm -f elf64 push.asm -o push.o -O0; ld -o push push.o
```

Push on 64 Bit

- Yes, it makes sense to specify what will be pushed

→ byte / word / dword

- Yes, on 32Bit you can push 4 bytes
- You cannot push 8byte onto the stack at 64Bit
- You need to work around it
- Simple mov is enough, drawback is more bytecode

Push

```
BITS 64
global _start
_start:
xor rax, rax ; clear register
; place 8byte in register rax
mov rax, 0x4142434445464748
; push it on the stack
push rax
```

```
nasm -f elf64 push_mov.asm -o push_mov.o -O0; ld -o push_mov push_mov.o
```

Recap: Push on 64 Bit

- < 5 byte push:

→ byte / word / dword

- > 4 byte push:

```
mov rax, 0x4142434445464748
push rax
```

Syscall: write

- lets look into how to push strings on the stack
- print it to the current shell
- look up the syscall write – man 2 write
- systemcall from unistd._64.h

Syscall: Write

- ssize_t write(int fd, const void *buf, size_t count);
- syscall nr is 1 or 0x1
- 3 Arguments
- we don't care about the return value
- write to stdout (stdin/stdout/stderr – 0/1/2)
- string is pushed on the stack
- you need the length of the string

Registers – Syscall Arguments

Register	Purpose	Other
RAX	Syscall	Return Value!
RDI	1. Argument	
RSI	2. Argument	
RDX	3. Argument	
R10	4. Argument	
R8	5. Argument	
R9	6. Argument	

Push strings

How to place a string on the stack:

- terminate the string
- newline the string (0x0a)
- record the length
- convert string to hex
- print string backwards in hex
- split it into byte size of registers you use
- easy no?

Push strings

Short version board tools (all in one):

print a[::-1].encode('hex')

Well...long version with extra library loaded:

In [11]: print a

- shellcoding at hack4 in 2015 -

In [12]: print a[::-1]

- 5102 ni 4kcah ta gnidoclleh s -

convert it to hex:

import binascii

binascii.hexlify(a[::-1])

2d2035313032206e6920346b63616820746120676e69646f636c6c656873202d

Syscall: Write
(Warning: the string in the code might be different)

```
BITS 64
global _start
;section .text:
_start:
xor rax, rax ; clear register
xor rdi, rdi ; clear register
push rax    ; ends the string
mov rax, 0x0a2035313032206e ; trick to place 8byte on the stack
push rax                ; push it
mov rbx, 0x6920346b63616820 ; same same, but different
push rbx
mov rcx, 0x746120676e69646f
push rcx
mov rdx, 0x636c6c6568732020
push rdx
mov rsi,rsp     ; move address of stack pointer to our 2nd argument
xor rax, rax    ; clean the register
mov al,1        ; move syscall write into accumulator register
inc di          ; arg 1, increment xor'ed register to stdout
xor rdx, rdx
add dl,byte 32
syscall
mov al,60
xor rdi, rdi
syscall
```

```
0000000000400080 <_start>:
  400080:  48 31 c0                xor    rax,rax
  400083:  48 31 ff                xor    rdi,rdi
  400086:  50                      push   rax
  400087:  48 b8 6e 20 32 30 31    movabs rax,0xa2035313032206e
  40008e:  35 20 0a
  400091:  50                      push   rax
  400092:  48 bb 20 68 61 63 6b    movabs rbx,0x6920346b63616820
  400099:  34 20 69
  40009c:  53                      push   rbx
  40009d:  48 b9 6f 64 69 6e 67    movabs rcx,0x746120676e69646f
  4000a4:  20 61 74
  4000a7:  51                      push   rcx
  4000a8:  48 ba 20 20 73 68 65    movabs rdx,0x636c6c6568732020
  4000af:  6c 6c 63
  4000b2:  52                      push   rdx
  4000b3:  48 89 e6                mov    rsi,rsp
  4000b6:  48 31 c0                xor    rax,rax
  4000b9:  b0 01                   mov    al,0x1
  4000bb:  66 ff c7                inc    di
  4000be:  48 31 d2                xor    rdx,rdx
  4000c1:  80 c2 20                add    dl,0x20
  4000c4:  0f 05                   syscall
  4000c6:  b0 3c                   mov    al,0x3c
  4000c8:  48 31 ff                xor    rdi,rdi
  4000cb:  0f 05                   syscall
```

Recap

- How to push strings on the stack
- Backwards/Hex
- We cannot push 8 byte

→ use mov

- Remember the string terminator

Syscall: Execve

- int execve(const char *filename, char *const argv[],

  char *const envp[]);

- how to print it in hex backwards another method:

> a=”//bin/sh”

> print a[::-1].encode('hex')

> 68732f6e69622f2f

- syscall execve from unistd_64:

59 or 3Bh

Syscall: Execve

```
xor rax, rax

push    rax                          ; null terminator for the string

mov     rbx, XXXXXXXXXXX  ; //bin/sh backwards

push    rbx                          ;

mov     rdi, rsp                     ; move address from stack pointer to first
argument

push    rax

push    rdi                          ; actually we would not need this one

mov         rsi, rsp                 ; move the address to the 2nd argument

mov         rdx, rax                 ; no envp necessary

mov         al,X                     ; execve into rax

syscall
```

```
0:   48 31 c0            xor    rax,rax
3:   50                  push   rax
4:   48 bb …………          movabs
rbx,0xXXXXXXXXXXXXXX
B:   ……..
e:   53                  push   rbx
f:   48 89 e7            mov    rdi,rsp
12:   50                 push   rax
13:   57                 push   rdi
14:   48 89 e6           mov    rsi,rsp
17:   48 89 c2           mov    rdx,rax
1a:   b0 3b              mov    al,….
1c:   0f 05              syscall
```

Syscall: Execve

- gain a shell via it
- still same user privileges
- gaining a root shell needs us to use setuid syscall

Execve + Setuid

- Ok. Now setuid(0) call needs to be added
- You want to have r00t, don't you?

Execve + Setuid

```
xor rax, rax
push rax
pop rdi
add al,0x69
syscall

; add the execve
shellcode, here
```

```
<_start>:
48 31 c0 xor    rax,rax
50       push   rax
5f       pop    rdi
04 69    add    al,0x69
0f 05    syscall
```

Execve + Setuid

- Simple extra call, now a r00t shell. Easy as that.

Other important syscalls

- everything in regard of sockets
- setuid / setgid / seteuid / setegid
- open / close / read / write
- fork / clone / chdir
- strongly depends on what you want to do

Other Shellcodes

- Now, the real fun part starts here:

→ bindshells

→ reverse shells

→ encoders / crypters / polymorphism

→ password protection

- But not today – sorry ;)

Fin.