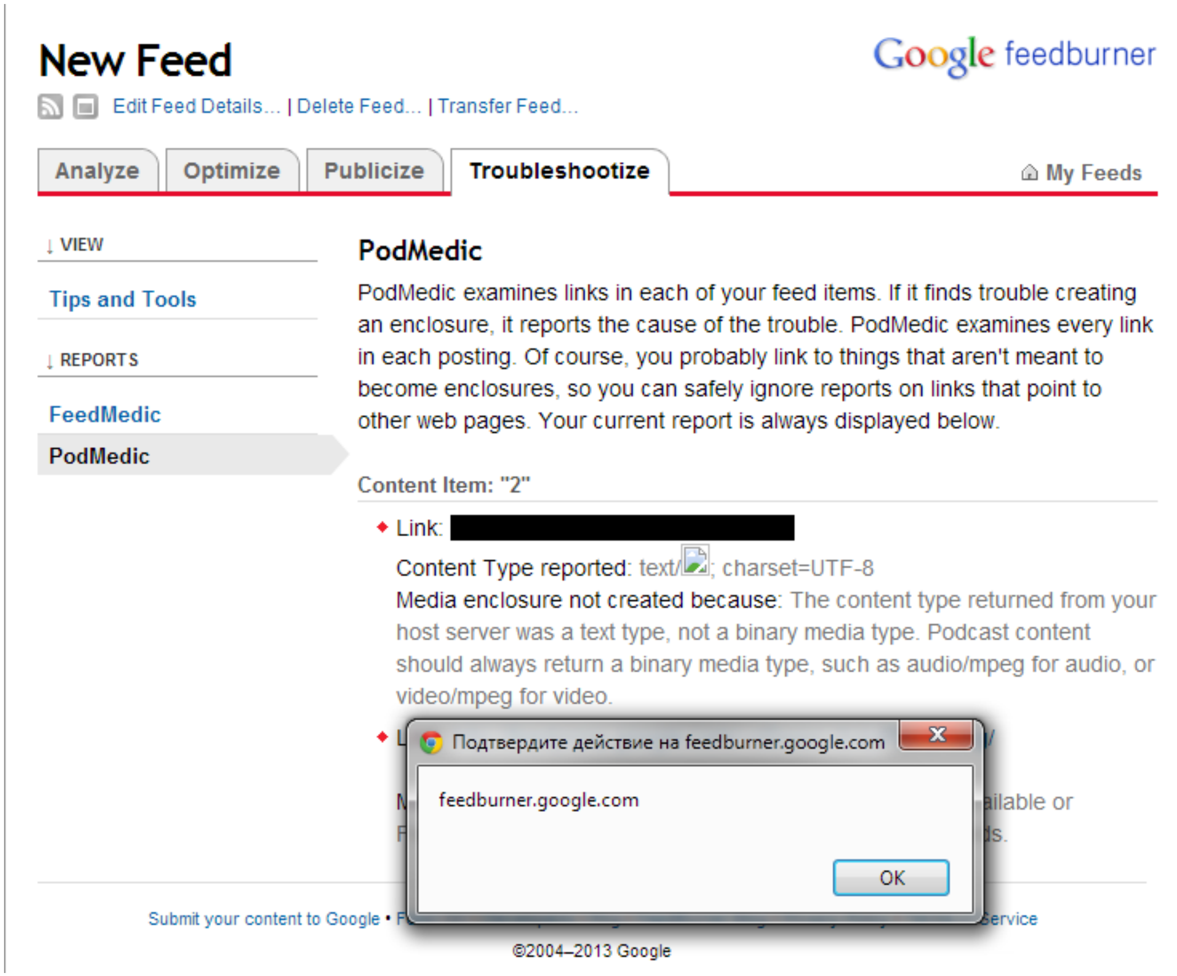
Buffer overflows – cont'd

Secure Programming
Lecture 7

In the news



http://paul-axe.blogspot.de/2014/01/two-stories-about-xss-on-google.html

In the news

includes/media/DjVu.php

```
@@ -180,9 +180,12 @@
 		$srcPath = $image->getLocalRefPath();
 		# Use a subshell (brackets) to aggregate stderr from both pipeline commands
 		# before redirecting it to the overall stdout. This works in both Linux and Windows XP.
-		$cmd = '(' . wfEscapeShellArg( $wgDjvuRenderer ) . " -format=ppm -page={$page}" .
-			" -size={$params['physicalWidth']}x{$params['physicalHeight']} " .
-			wfEscapeShellArg( $srcPath );
+		$cmd = '(' . wfEscapeShellArg(
+			$wgDjvuRenderer,
+			"-format=ppm",
+			"-page={$page}",
+			"-size={$params['physicalWidth']}x{$params['physicalHeight']}",
+			$srcPath );
 		if ( $wgDjvuPostProcessor ) {
 			$cmd .= " | {$wgDjvuPostProcessor}";
 		}
```

includes/media/ImageHandler.php

```
@@ -93,6 +93,7 @@
 		if ( !isset( $params['page'] ) ) {
 			$params['page'] = 1;
 		} else {
+			$params['page'] = intval( $params['page'] );
 			if ( $params['page'] > $image->pageCount() ) {
 				$params['page'] = $image->pageCount();
 			}
```

https://git.wikimedia.org/commitdiff/mediawiki%2Fcore.git/9e0b52b14537a4238239a6e8a2319fe3838fa409

Procedure call

```
#include <stdio.h>

int foo(int a, int b) {
    int c = 42;

    return a * b + c;
}

int main(int argc, char **argv) {
    int c = 0;

    c = foo(1, 2);

    printf("%d\n", c);

    return 0;
}
```

2
1
Saved IP
Saved BP
42

Under the hood

```
(gdb) disass main
Dump of assembler code for function main:
   0x080483fd <+0>:     push   %ebp
   0x080483fe <+1>:     mov    %esp,%ebp
   0x08048400 <+3>:     and    $0xfffffff0,%esp
   0x08048403 <+6>:     sub    $0x20,%esp
   0x08048406 <+9>:     movl   $0x0,0x1c(%esp)
   0x0804840e <+17>:    movl   $0x2,0x4(%esp)
   0x08048416 <+25>:    movl   $0x1,(%esp)
   0x0804841d <+32>:    call   0x80483e4 <foo>
   0x08048422 <+37>:    mov    %eax,0x1c(%esp)
   0x08048426 <+41>:    mov    $0x8048520,%eax
   0x0804842b <+46>:    mov    0x1c(%esp),%edx
   0x0804842f <+50>:    mov    %edx,0x4(%esp)
   0x08048433 <+54>:    mov    %eax,(%esp)
   0x08048436 <+57>:    call   0x8048300
<printf@plt>
   0x0804843b <+62>:    mov    $0x0,%eax
   0x08048440 <+67>:    leave
   0x08048441 <+68>:    ret
End of assembler dump.
```

0x2
0x1
0x8048422

Under the hood

```
(gdb) disass foo
Dump of assembler code for function foo:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x10,%esp
   0x080483ea <+6>:     movl   $0x2a,-0x4(%ebp)
   0x080483f1 <+13>:    mov    0x8(%ebp),%eax
   0x080483f4 <+16>:    imul   0xc(%ebp),%eax
   0x080483f8 <+20>:    add    -0x4(%ebp),%eax
   0x080483fb <+23>:    leave
   0x080483fc <+24>:    ret
End of assembler dump.
```

0x2
0x1
0x8048422
0xbffff1a8
0x2a

Let’s run it

```
(gdb) br foo
Breakpoint 1 at 0x80483ea: file procedure.c, line 4.
(gdb) r
Starting program: procedure foo

Breakpoint 1, foo (a=1, b=2) at procedure.c:4
4           int c = 42;
(gdb) stepi
6           return a * b + c;
(gdb) x/8wx $ebp -4
0xbffff174: 0x0000002a 0xbffff1a8 0x08048422 0x00000001
0xbffff184: 0x00000002 0x002d8ff4 0x00166225 0x0011f270
```

0x2
0x1
0x8048422
0xbffff1a8
0x2a

Returning

```
(gdb) x/i $eip
=> 0x80483fb <foo+23>:  leave
(gdb) info r ebp esp
ebp            0xbffff178       0xbffff178
esp            0xbffff168       0xbffff168
(gdb) si
(gdb) info r ebp esp
ebp            0xbffff1a8       0xbffff1a8
esp            0xbffff17c       0xbffff17c
(gdb) x/i $eip
=> 0x80483fc <foo+24>:  ret
(gdb) si
0x08048422 in main (argc=2, argv=0xbffff244) at
procedure.c:12
(gdb) info r ebp esp eip
ebp            0xbffff1a8       0xbffff1a8
esp            0xbffff180       0xbffff180
eip            0x8048422        0x8048422 <main+37>
```

0x2
0x1

BUFFER OVERFLOWS

Vulnerable program

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

void vulnerable(char* param)
{
    char buffer[100];
    strcpy(buffer, param);
}

int main(int argc, char** argv)
{
    vulnerable(argv[1]);
    printf("OK\n");

    return 0;
}
```

Let’s crash

```
06-buffer-overflows$ ./vuln test
OK

marco@ubuntu:$ ./vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

Let’s crash

```
$ gdb vuln
(gdb) r
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA
Starting program: vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info r esp ebp eip
esp            0xbffff0f0     0xbffff0f0
ebp            0x41414141     0x41414141
eip            0x41414141     0x41414141
(gdb) x/20wx $esp -32
0xbffff0d0:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff0e0:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff0f0:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff100:     0x41414141     0x41414141     0x41414141     0x41414141
0xbffff110:     0x41414141     0x41414141     0x41414141     0x41414141
```

To reproduce

Disable defense mechanisms!

```
$ gcc -fno-stack-protector \
      -U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 \
      vuln.c -o vuln
```

(We'll see these in detail in a few lectures.)

Smashing the stack

Key idea: Overwrite a pointer with the address of code we want to execute

1. locate a pointer that (eventually) will be copied to the EIP register or that points to data that will be copied to the EIP
   - function pointers (on the stack, heap, BSS, …)
   - saved EBP
   - procedure return address
   - entry in the GOT
   - jmp_buf
2. overwrite the pointer with our value

Smashing the stack

- A procedure contains a local buffer variable allocated on the stack
- The procedure copies user-controlled data to the buffer without verifying that the data size is smaller than the buffer
- The user data overwrites the other variables on the stack, up to the *return address* saved in the function frame
- When the procedure returns, the program fetches the return address from the stack and copies it to the EIP register
- Since we can control the return address, we can jump to an address of our choice

Where do we jump to?

Address inside a buffer whose content is under user control

- PRO: works for remote attacks
- CON: the attacker needs to know the address of the buffer
- CON: the memory page containing the buffer must be executable

Where do we jump to?

Address of an environment variable

- PRO: easy to implement, works with tiny buffers
- CON: only for local exploits
- CON: some programs clean the environment
- CON: the environment must be executable

Where do we jump to?

- Address of a function inside the program
- PRO: works for remote attacks
- PRO: does not require an executable stack
- CON: need to find the right code
- CON: one or more fake frames must be put on the stack

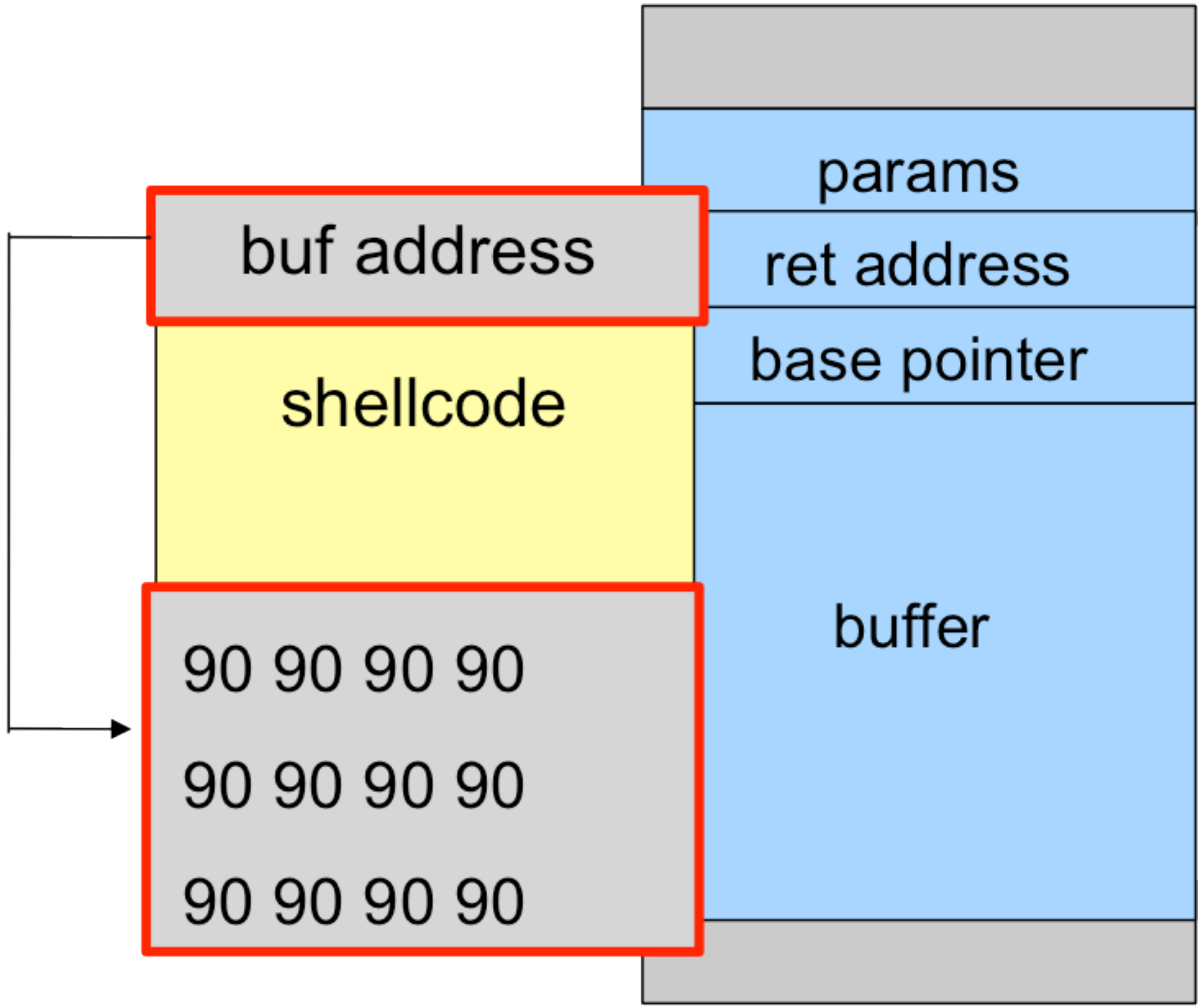
Jumping into the buffer

- The buffer that we are overflowing is usually a good place to put the code (shellcode) that we want to execute
- The buffer is somewhere on the stack, but in most cases the exact address is unknown
- The address must be precise: jumping one byte before or after would just make the application crash
- On the local system, it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine: any change to the environment variables affects the stack position

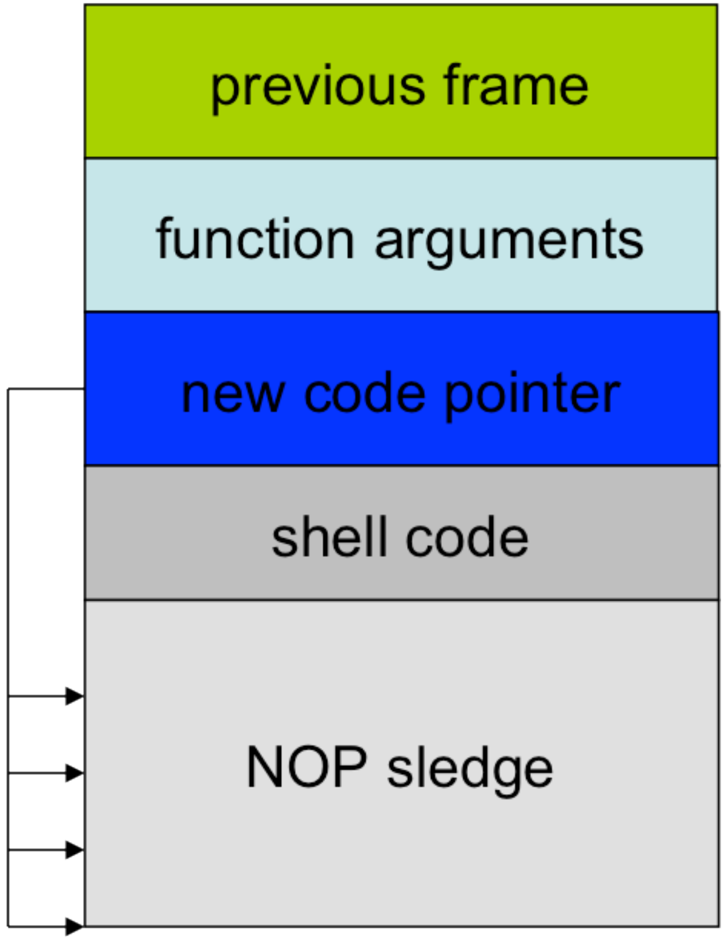
Solution #1: NOP sled

- A sled is a “landing area” in front of the shellcode
  - work arounds the problem of finding the right shellcode address
- Must be created in a way such that whenever the program jumps into it:
  - it always finds a valid instruction
  - it always reaches the end of the sled and the beginning of the shellcode
- The simplest sled is a sequence of NOP instructions
  - single byte instruction (0x90) that does not do anything
  - more complex sleds possible

Assembling the malicious input



Assembling the malicious input



any return address into the NOP sled succeeds

Solution #2: register-based jump

- Find a register that points to the buffer (or somewhere into it)
  - ESP
  - EAX (return value of a function call)

- Locate an instruction that jumps/calls using that register
  - can also be in one of the libraries
  - does not need to be a real instruction, just look for the right sequence of bytes (jmp *$esp = 0xFF 0xE4)

- Overwrite the return address with the address of that instruction

(We will talk about address space randomization)

Solution #3: heap spraying

- We will talk about this later

Some history

Morris worm (1988): overflow in fingerd

- 6,000 machines infected (10% of the Internet)
- Internet had to be switched off
- CERT is created

CodeRed (2001): overflow in MS IIS server

- 300,000 machines infected in 14 hours

SQL Slammer (2003): overflow in MS SQL server

- attack: 1 UDP packet
- 75,000+ machines infected in 10 minutes

In 2003, around 75% of the vulnerabilities were buffer overflows x86

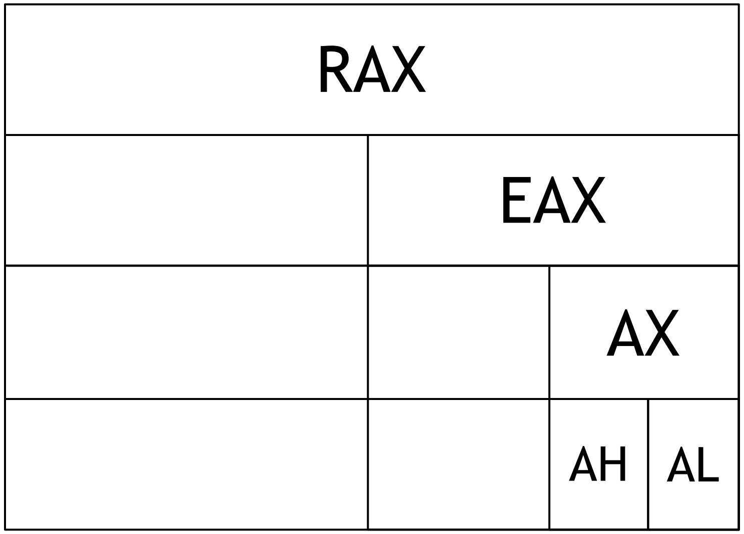
ASSEMBLY REFRESHER

Registers

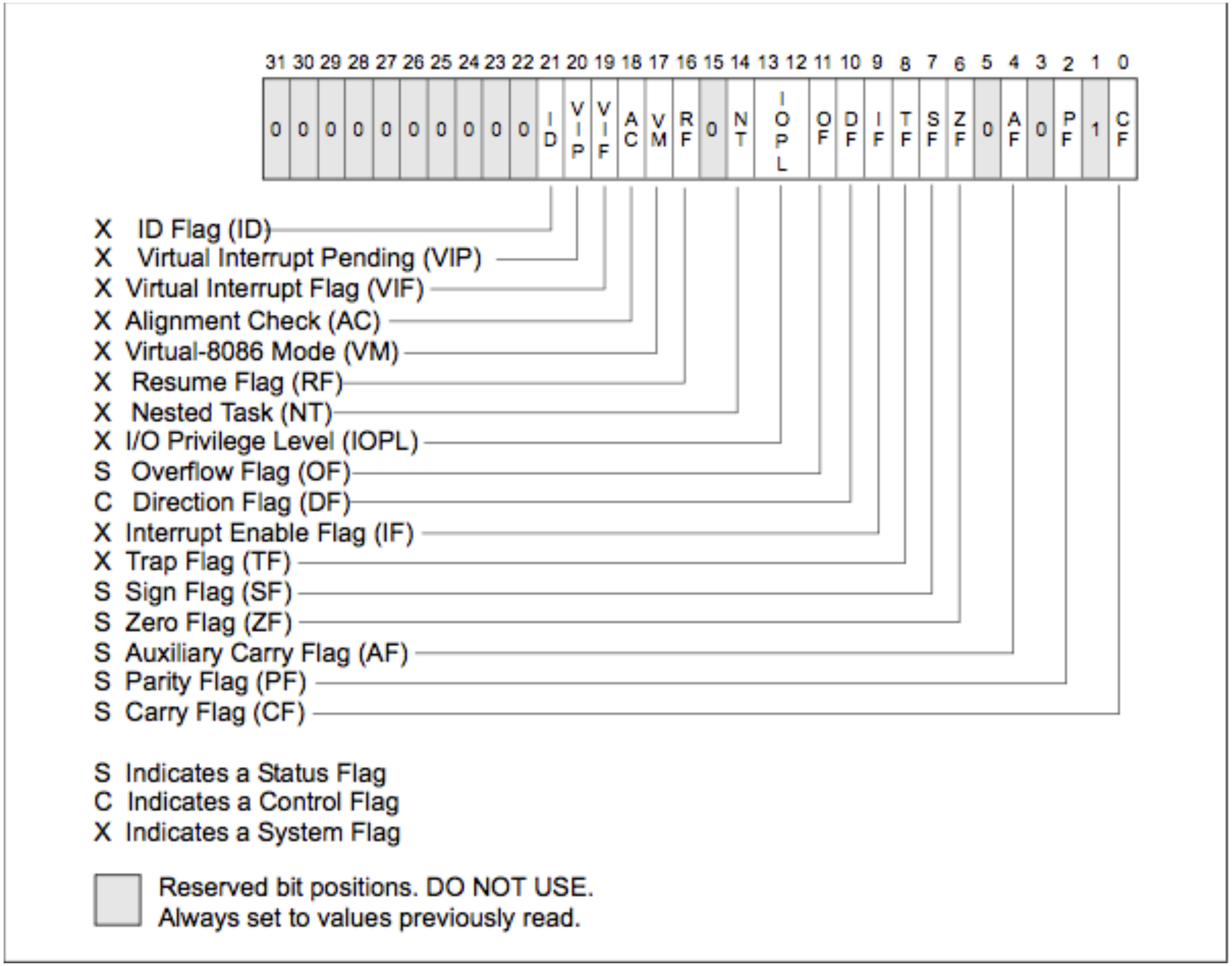
General purpose registers:

- AX (accumulator)
- BX (base)
- CX (counter)
- DX (data)
- SP (stack pointer)
- BP (stack base pointer)
- SI (source index)
- DI (destination index)

- Can be accessed in 5 modes: 64, 32, 16, 8 bit (LSB and MSB)



Flags register



Instruction pointer register

The instruction pointer (EIP) register

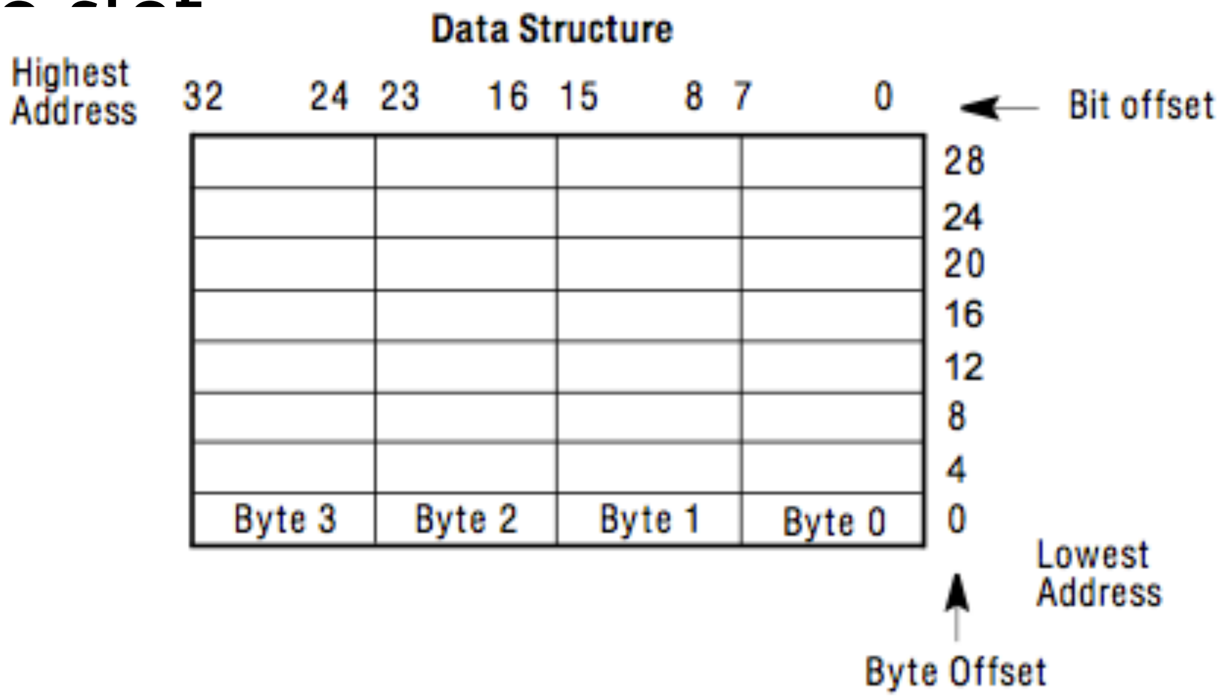
- Contains the offset in the current code segment for the next instruction to be executed

Cannot be accessed directly by software:

- It is advanced from one instruction boundary to the next in straight-line code
- moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

Endiannes

- x86 is a little-endian architecture: the least significant byte goes into the lowest value slot



Data transfer instructions

- `mov src, dest`

  Copy the source operand (immediate, register, memory) into the destination operand (register, memory)

- `lea src, dest`

  Calculate the address of the source operand (immediate, register, memory) and load it into the destination operand (register)

  (Used also for general-purpose arithmetic operations)

Control flow instructions

- `test arg1, args2`
  Perform a bit-wise logical AND on arg1, arg2 and set ZF (zero), SF (sign), and PF (parity) flags
- `cmp arg2, arg1`
  Perform a signed subtraction of arg2 from arg1 and set ZF, SF, PF accordingly
- `jmp loc`
  Unconditional jump
- `jcc loc`
  Conditional jump (je, jne, jg, jge, ja, jae, jl, jle, jb, jbe, jz, jnz, etc.)

Function calls

- `call proc`

  Push the address of the next opcode on the stack and jump to specified address

- `ret [val]`

  Load the next value on the stack into EIP, pop the specified number of bytes off the stack (none if not specified)

Arithmetic instructions

- `add src, dst`

  Add the source operand to destination operand and store the result in the destination operand
- `sub src, dst`
- `mul arg`

  Multiply arg by the value of the corresponding byte-length in AX
- `div arg`
- `inc arg`
- `dec arg`
- …

Logic instructions

- `and src, dest`

  Perform a bit-wise AND of the operands and store the result in the destination operand.
- `or src, dest`
- `xor src, dest`
- `not arg`

  Perform a bit-wise inversion of arg

Stack instructions

- `push arg`

  Decrement the stack pointer and store the value of the argument in the location pointed to by SP
- `pop arg`

  Load the data stored in the location pointed to by SP into the argument and then increments the SP

References

- GDB: The GNU Project Debugger
- Intel 64 and IA-32 Architectures Software Developer's Manual
- Aleph One, Smashing The Stack For Fun And Profit, Phrack 49, 1996
- spoonm, Recent Shellcode Developments, ReCon 2005