IndianZ

Coding

Coding is about creating or modifying code – a necessary survival skill in the world of IT security.

December 2010

Haftung + Verantwortung

• Die in dieser Präsentation beschriebenen Techniken können auch für kriminelle Zwecke verwendet werden

• Verantwortungsvoller Umgang mit diesem Wissen wird vorausgesetzt

• IndianZ übernimmt KEINERLEI Haftung bei der legalen oder illegalen Anwendung dieses Wissens

Agenda

- Introduction
- C
- Computer Memory
- Intel Processors
- ASM
- Perl
- Python
- Shellscripting
- More Tools

Begrifflichkeit

- C = computer programming language developed in 1972 by Dennis Ritchie (Bell)
  - GCC = GNU C Compiler | GDB = GNU DeBugger
- ASM = family of low-level languages for programming computers and microprocessors
- Perl = high-level, interpreted, dynamic programming language, developed in 1987 by Larry Wall (NASA)
- Python = high-level programming language (code readability), developed in 1991 by Guido van Rossum (CWI)
- Bash = shell scripting language, developed in 1987 von Brian Fox and extended in 1990 by Chet Ramey

Introduction

- Programmierer
  - Ordnung, Schönheit, Methode, Grösse, Inside-the-box, Verteidigung, Geld verdienen
  - Zeit: Time to market !?

- Hacker
  - Unordnung, Fuzzing, Quick'n'Dirty, Outside-the-box, Angriff, Bluffen
  - Zeit: soviel es braucht ;)

Introduction

- Problem solving process
  - 1 Define the problem
  - 2 Distill the problem down to byte-sized chunks
  - 3 Develop pseudo-code
  - 4 Group like components into modules
  - 5 Translate to a programming language
  - 6 Debug errors (Syntax)
  - 7 Runtime errors
  - 8 Test the program
  - 9 Implement production

Introduction

- First line of scripts = magic line = shebang
- Examples

  #!/usr/bin/perl — Perl

  #!/usr/bin/perl -w — Perl with warnings

  #!/usr/bin/python — Python

  #!/usr/bin/env python — Python over env

  #!/bin/sh — Sh

  #!/bin/csh — Csh

  #!/bin/bash — Bash

Understand C

C

Understand C

- **C-Constructs**
  - **main()**

    <optional return value type> main (optional argument) {

    <optional procedure statements or function calls>

    }
  - **command line arguments**

    <optional return value type> main(int argc, char * argv[]){

Understand C

- C-Constructs
  - functions

    <optional return value type> function name (<optional function argument>){

    }
  - first line of function = signature

    <optional variable to store the returned value 0>function name (arguments if called for by the function signature);

Understand C

- **C-Constructs**
  - **Variables overview**

Variable Type	Use	Typical Size
int	Stores signed integer values such as 314 or –314	4 bytes for 32-bit machines 2 bytes for 16-bit machines
float	Stores signed floating point numbers; for example, –3.234	4 bytes
double	Stores large floating point numbers	8 bytes
char	Stores a single character such as "d"	1 byte

Understand C

- C-Constructs
  - variables
    - <variable type> <variable name> <optional initialization starting ewith “=”>;
    - Example: int a= 0;
    - Example: x=x+1;
  - destination = where final output is stored
    - destination = source <with optional operators>

Understand C

- C-Constructs (bundled with libc)
  - printf

    printf(<string>);

    printf(<format string>), <list of variables/values>);

Format Symbol	Meaning	Example
\n	Carriage return/new line	printf("test\n");
%d	Decimal value	printf("test %d", 123);
%s	String value	printf("test %s", "123");
%x	Hex value	printf("test %x", 0x123);

Understand C

- **C-Constructs (bundled with libc)**
  - **scanf**

    scanf(<format string>, <list of variables/values>);
    - **Example: scanf(“%d”, &number);**
  - **strcpy/strncpy**
  - **strcpy most dangerous command in C**

    strcpy(<destination>, <source>);

    strncpy(<destination>, <source>, <width>);

Understand C

- **C-Constructs**
  - **For (use < not <=, off-by-one ;)**

```
for(<beginning value>; <test value>; <change value>){
}
```

  - **Example:**

```
for (i=0; i<10; i++){
  printf(“%d”, i);
}
```

Understand C

- **C-Constructs**
  - **while**

```
while(<conditional test>){
  <statement>;
}
```

Understand C

- **C-Constructs**
  - **if/else**

```
if(<condition>){
  <statements to exec when condition is met>
} <else>{
  <statements to exec when condition is not met>;
}
```

Understand C

- Comments
  - 1: // omits rest of line
  - 2: /* */ omits multiple lines
- Example Program

```
//hello.c                     //program name comment
#include <stdio.h>            //screen printing
main ( ) {                    //required main function
    printf("Hello haxor");    //simply say hello
}                             //exit program
```

Compiling with GCC

- **gcc -o object object.c**

- **GCC Flags**

Option	Description
-o <filename>	The compiled binary is saved with this name. The default is to save the output as a.out.
-S	The compiler produces a file containing assembly instructions; saved with a .s extension.
-ggdb	Produces extra debugging information; useful when using GNU debugger (gdb).
-c	Compiles without linking, produces object files with a .o extension.
-mpreferred-stack-boundary=2	A useful option to compile the program using a DWORD size stack, simplifying the debugging process while you learn.

Understand C

- //meet.c
- #include <stdio.h>   // screen printing
- greeting(char *temp1,char *temp2){   // greeting function
-    char name[400];   // string variable: name
-    strcpy(name, temp2);   // cp function arg to name
-    printf("Hello %s %s\n", temp1, name); //print greeting
- }
- main(int argc, char * argv[]){   // note arg format
-    greeting(argv[1], argv[2]);   //call function title+name
-    printf("Bye %s %s\n", argv[1], argv[2]); //say "bye"
- }   //exit program

Debugging with GDB

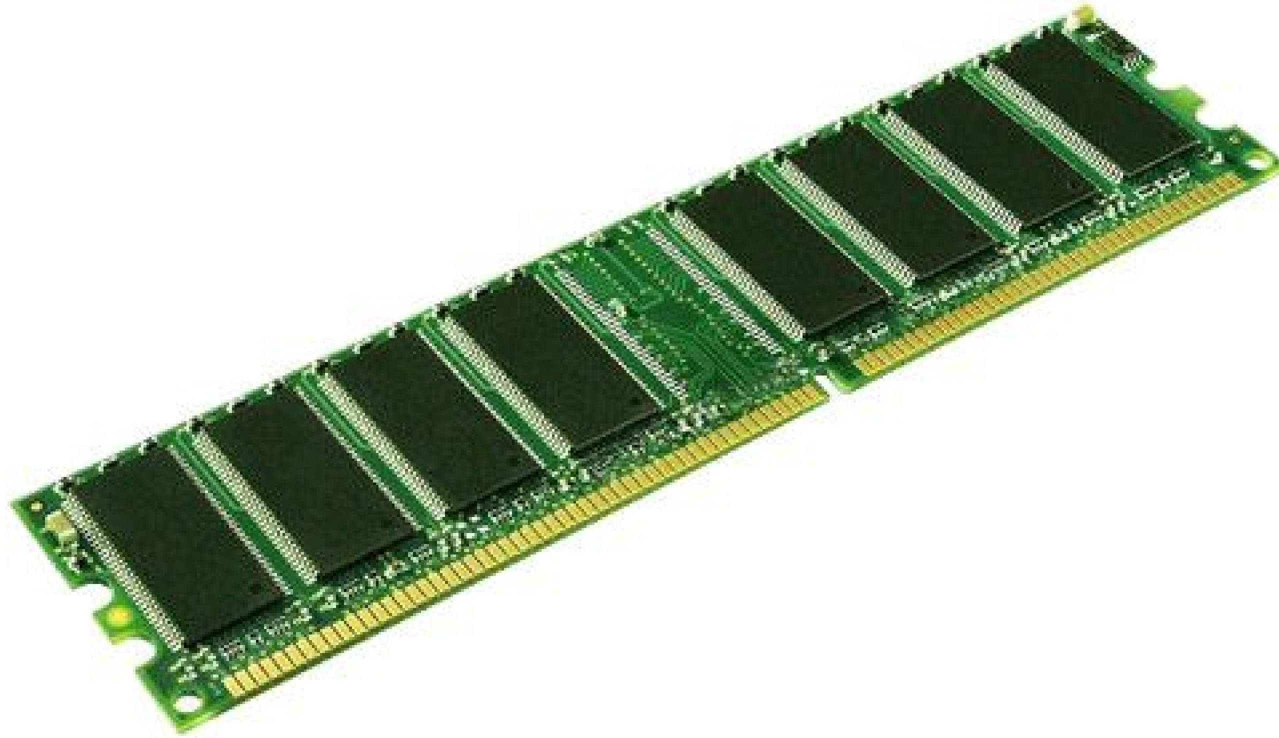
- **GDB Commands**

Command	Description
b *function*	Sets a breakpoint at *function*
b *mem	Sets a breakpoint at absolute memory location
info b	Displays information about breakpoints
delete b	Removes a breakpoint
run <args>	Starts debugging program from within gdb with given arguments
info reg	Displays information about the current register state
stepi or si	Executes one machine instruction
next or n	Executes one function
bt	Backtrace command which shows the names of stack frames
up/down	Moves up and down the stack frames

Debugging with GDB

- GCC for GDB
  - gcc -ggdb -mpreferred-stack-boundary=2 -o meet meet.c
  - gdb -q meet
    - run
    - b main
    - ...
- set disassembly-flavor <intel/att>
- disassemble <function name>

Computer Memory



Computer Memory

- **Bit's und Bytes**
  - **0 or 1= 1bit**
  - **4 bit (0000 bis 1111) / (0-15) = 1 nibble**
  - **8 bit (0 – 2^8 -1) / 0-255) = 1 byte**
  - **2 bytes (0 – 2^{16} -1) / (0-65535) = 1 word**
  - **2 words (0 – 2^{32} -1) / (0-4'294'967'295) = 1 double word**

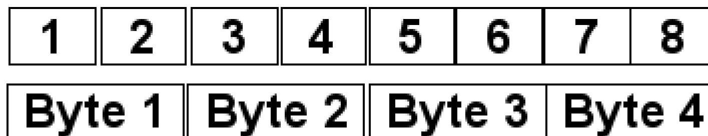
Computer Memory

- RAM
  - Random Access Memory
  - Volatile (lost when power off)
  - X86 = 32bit
  - Max limit: 4'294'967'295 bytes
- Registers
  - Special form of embedded memory on CPU

Computer Memory

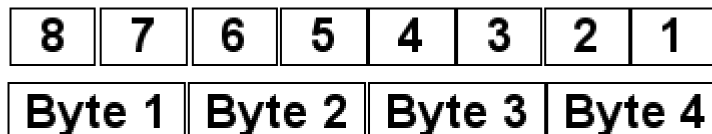
- **Big Endian (Motorola, SPARC, 64bit)**
- **Low-order bytes written first**

Höherwertiges Byte Niederwertiges Byte



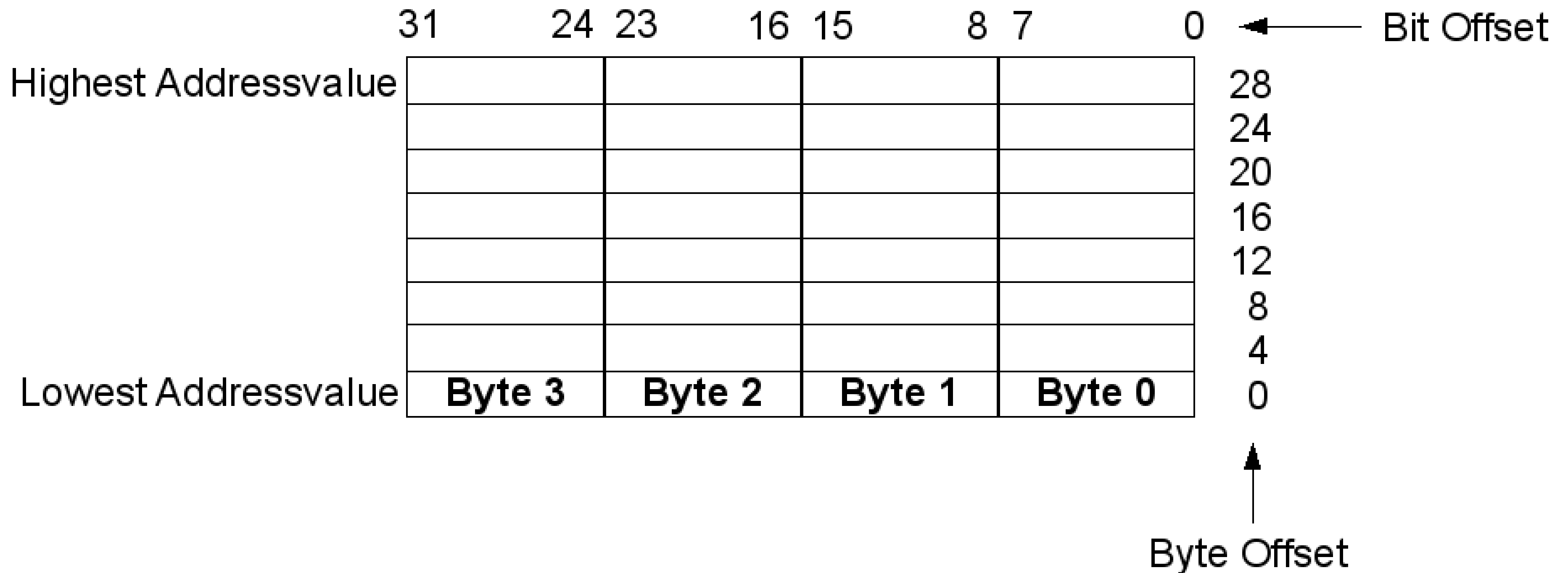
- **Little Endian (Intel, 32bit)**
- **High-order bytes written first**

Niederwertiges Byte Höherwertiges Byte



Computer Memory

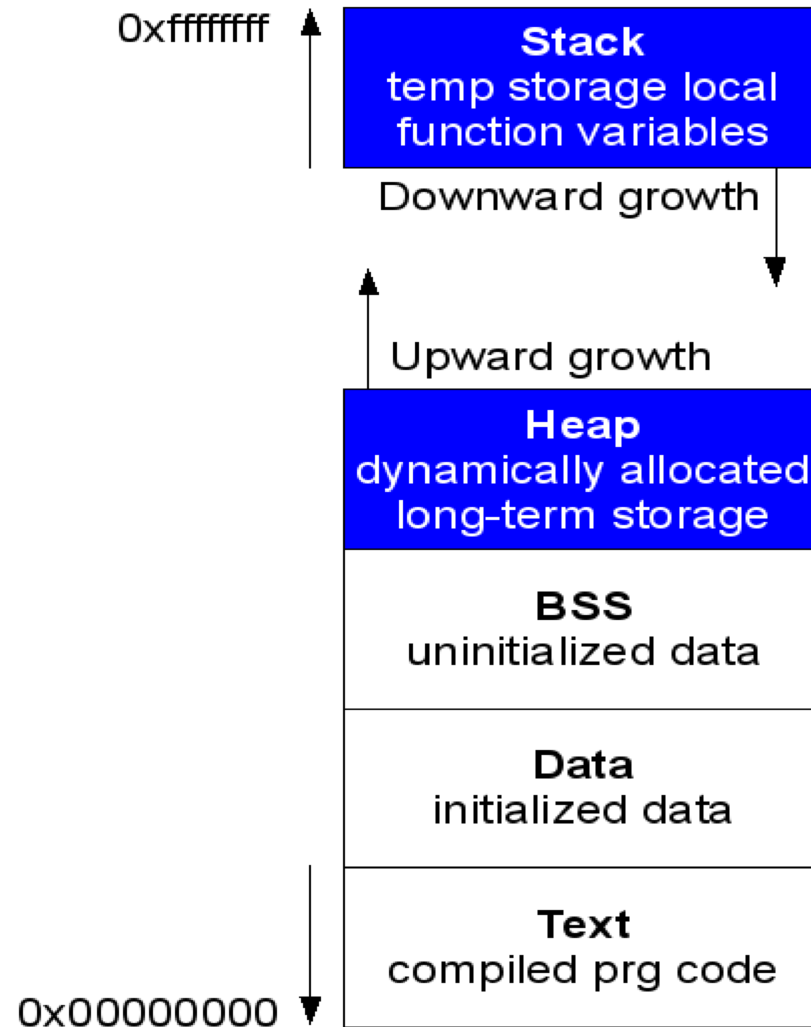
- **Little Endian (IA-32/x86)**
- **Ausgehend vom Least Significant Bit**



Computer Memory

- **Segmentation**
  - **.text = machine instructions, read only, segfaults, size = fixed at runtime when process is loaded**
  - **.data = global initialized variables (int a = 0;), size fixed at runtime**
  - **.bss = global non-initialized variables (int a;), size fixed at runtime**
  - **heap = dynamically allocated variables (int i = malloc (sizeof (int));), grows from lower to higher addressed memory**
  - **stack = keeps track of function calls (recursively), grows from higher to lower addressed memory, contains local variables**
  - **env = stores a copy of system-level variables (path, shell, hostname)**

Computer Memory



Computer Memory

- Process Memory Layout

- Buffers
  - Storage place used to receive and hold data until handled by process, allocating .data/.bss
- Strings
  - Continuous arrays of character data in memory, referenced by address of first character, termination by null (\0 in C)

Computer Memory

- Pointers
  - Special pieces of memory, which hold address of other pieces of memory
  - Saved in 4 bytes (32bits)
  - Example:

```
char * str;  // read, gives 4 bytes pointer to char var, bss
int * point1;    // read, 4 bytes pointer to int var
```

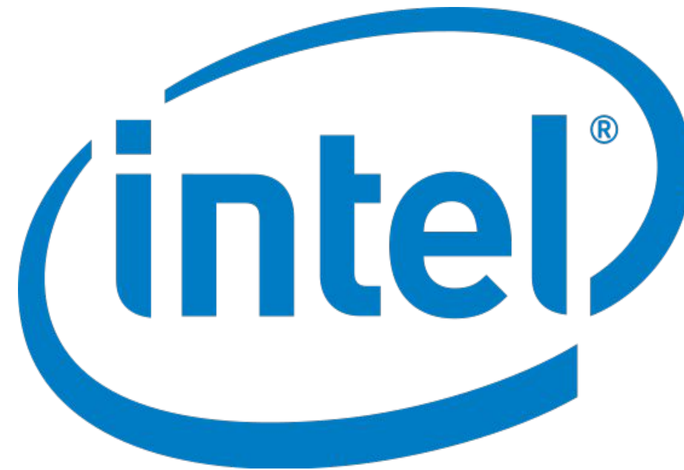
  - Dereference with * symbol

```
printf("%d", *point1);
```

Computer Memory

```
/* memory.c */      // this comment holds the program name
   int index = 5;     // integer stored in data (initialized)
   char * str;          // string stored in bss (uninitialized)
   int nothing;        // integer stored in bss (uninitialized)
void funct1(int c){   // bracket starts function1 block
   int i=c;                // stored in the stack region
   str = (char*) malloc (10 * sizeof (char));  // Reserve on heap
   strncpy(str, "abcde", 5);  //copy 5 chars "abcde" into str
}                                     //end of function1
main (){                              //the required main function
   funct1(1);                          //main calls function1 with argument
}                                     //end of the main function
```

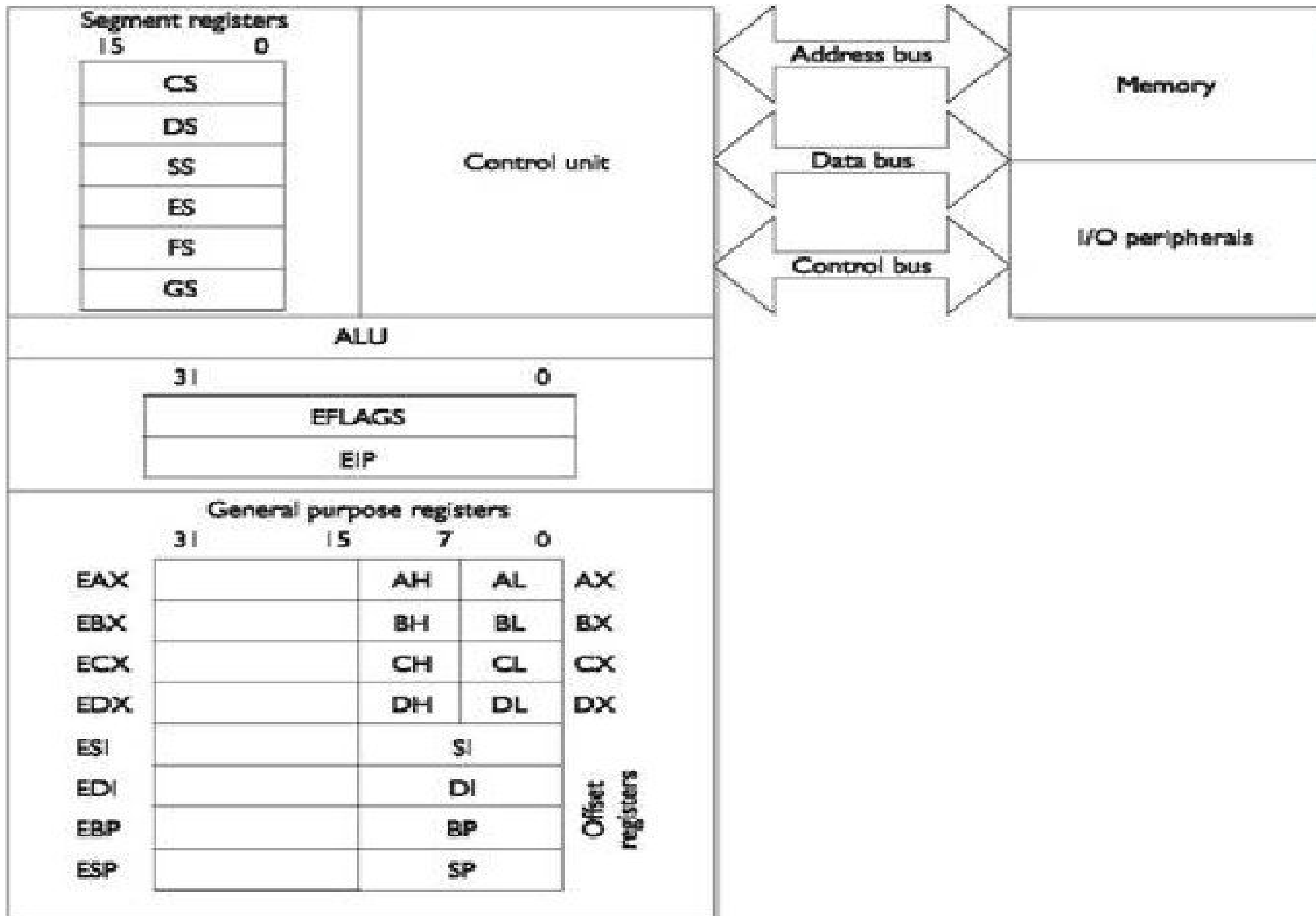
Intel Processors



Intel Processors

Register Category	Register Name	Purpose
General registers	EAX, EBX, ECX, EDX	Used to manipulate data
	AX, BX, CX, DX	16-bit versions of the preceding entry
	AH, BH, CH, DH, AL, BL, CL, DL	8-bit high and low order bytes of the previous entry
Segment registers	CS, SS, DS, ES, FS, GS	16-bit, holds the first part of a memory address; holds pointers to code, stack, and extra data segments
Offset registers		Indicates an offset related to segment registers
	EBP (extended base pointer)	Points to the beginning of the local environment for a function
	ESI (extended source index)	Holds the data source offset in an operation using a memory block
	EDI (extended destination index)	Holds the destination data offset in an operation using a memory block
	ESP (extended stack pointer)	Points to the top of the stack
Special registers		Only used by the CPU
	EFLAGS register; the key flags to know are: ZF=zero flag; IF=Interrupts; SF=sign	Used by the CPU to track results of logic and the state of processor
	EIP (extended instruction pointer)	Points to the address of the next instruction to be executed

Intel Processors



Understand Assembly

Understand Assembly

- AT&T = GNU Assembler (gas) in GCC-Suite
- NASM = Netwide Assembler

- NASM: CMD <dest>, <src> <; comment>
- AT&T: CMD <src>, <dest> <# comment>
- AT&T uses a % before registers, NASM not
- AT&T format uses a $ before literal values, NASM not
- AT&T handles memory references differently than NASM

Understand Assembly

- **mov**

NASM Syntax	NASM Example	AT&T Example
mov <dest>, <source>	mov eax, 51h ;comment	movl $51h, %eax #comment

- **add/sub**

NASM Syntax	NASM Example	AT&T Example
add <dest>, <source>	add eax, 51h	addl $51h, %eax
sub <dest>, <source>	sub eax, 51h	subl $51h, %eax

- **push/pop**

NASM Syntax	NASM Example	AT&T Example
push <value>	push eax	pushl %eax
pop <dest>	pop eax	popl %eax

Understand Assembly

- **xor**

NASM Syntax	NASM Example	AT&T Example
xor <dest>, <source>	xor eax, eax	xor %eax, %eax

- **jne/je/jz/jnz/jmp**

NASM Syntax	NASM Example	AT&T Example
jnz <dest> / jne <dest>	jne start	jne start
jz <dest> /je <dest>	jz loop	jz loop
jmp <dest>	jmp end	jmp end

- **call/ret**

NASM Syntax	NASM Example	AT&T Example
call <dest>	call subroutine1	call subroutine1
ret	ret	ret

Understand Assembly

- **inc/dec**

NASM Syntax	NASM Example	AT&T Example
inc <dest> dec <dest>	inc eax dec eax	incl %eax decl %eax

- **lea**

NASM Syntax	NASM Example	AT&T Example
lea <dest>, <source>	lea eax, [dsi +4]	leal 4(%dsi), %eax

- **int**

NASM Syntax	NASM Example	AT&T Example
int <val>	int 0x80	int $0x80

Understand Assembly

- **Addressing Mode**

Addressing Mode	Description	NASM Examples
Register	Registers hold the data to be manipulated. No memory interaction. Both registers must be the same size.	mov ebx, edx add al, ch
Immediate	Source operand is a numerical value. Decimal is assumed; use **h** for hex.	mov eax, 1234h mov dx, 301
Direct	First operand is the address of memory to manipulate. It's marked with brackets.	mov bh, 100 mov[4321h], bh
Register Indirect	The first operand is a register in brackets that holds the address to be manipulated.	mov [di], ecx
Based Relative	The effective address to be manipulated is calculated by using **ebx** or **ebp** plus an offset value.	mov edx, 20[ebx]
Indexed Relative	Same as Based Relative, but **edi** and **esi** are used to hold the offset.	mov ecx,20[esi]
Based Indexed-Relative	The effective address is found by combining based and indexed modes.	mov ax, [bx][si]+1

Understand Assembly

- Assembly File
  - .model = indicates size of .data and .text
  - .stack = marks beginning of stack segment and indicates size of stack
  - .data = mark sbeginning of data segment, defines variables (initialized/uninitialized)
  - .text = holds program commands
- Assembly
  - nasm -f elf hello.asm
  - ld -s -o hello hello.o

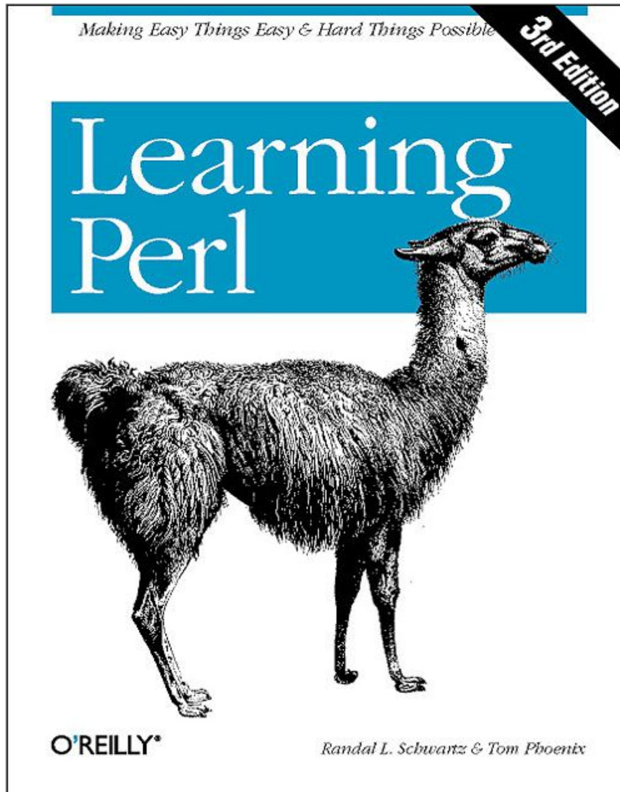
Understand Assembly

```
section .data                       ;section declaration
msg db "Hello, haxor!",0xa  ;our string with a carriage return
len equ $ - msg                     ;length of our string, $ means here
section .text         ;mandatory section declaration
                      ;export the entry point to the ELF linker or
    global _start     ;loaders conventionally recognize
                      ; _start as their entry point
_start:
```

Understand Assembly

```
                  ;now, write our string to stdout
                  ;notice how arguments are loaded in reverse
mov edx,len       ;third argument (message length)
mov ecx,msg       ;second argument (pointer to message to write)
mov ebx,1         ;load first argument (file handle (stdout))
mov eax,4         ;system call number (4=sys_write)
int 0x80          ;call kernel interrupt and exit
mov ebx,0         ;load first syscall argument (exit code)
mov eax,1         ;system call number (1=sys_exit)
int 0x80          ;call kernel interrupt and exit
```

Understand Perl



Understand Perl

- **Practical Extraction and Reporting Language or Pathologically Eclectic Rubbish Lister ;)**
- **Fokus auf Files, Strings, and Regular expressions**
- **Quick Text Processing and Portability**

- **perl file.pl**
- **#!/usr/bin/perl -w**
  require 5.004;
- **# comments #**

Understand Perl

- **Scalar Variables**
  - **$a = 17;**
  - **$b = 0x11;   # Hexadecimal (17 in decimal)**
  - **$c = 021;   # Octal   (17 in decimal)**
  - **$d = 0b10001; # Binary   (17 in decimal)**
  - **$f = 3.142;   # Floating point**
  - **$a = $a + 1; # Add 1 to variable $a**
  - **$a += 1;   # Add 1 to variable $a**
  - **$a++;   # Add 1 to variable $a**

Understand Perl

- **Scalar Variables**
  - **$b = $b * 10; # Multiply variable $b by 10;**
  - **$b *= 10;   # Multiply variable $b by 10;**
- **Arithmetic operators**
  - **** Exponentiation** — **%   Modulo division**
  - **++ Auto increment** — **-- Auto decrement**
  - **<   Numeric less than** — **>   Numeric greater than**
  - **== Numeric equality** — **!= Numeric inequality**
  - **<= less than or equal to** — **>= greater than or equal to**
  - **<=> Numeric compare: Returns -1 0 1**

Understand Perl

- **Scalar Variables**
  - **$a = 'Number of: '; # No interpolation with 'single quotes'**
  - **$b = "$a$c\n";   # Interpolation (variable substitution)**
  - **print $b;   # This makes "Number of: 17\n" appear**
  - **print $a, $c, "\n";   # As does this**
  - **print "That's all\n";   # No commas = a list of one element**
- **String Operators**
  - **lt = less than   gt = greater than**
  - **le = less than or equal to   ge = greater than or equal to**
  - **cmp String compare: Returns -1 0 1**

Understand Perl

- **Logic and Truth**
  - **0; # Integer zero** **0.0; # Decimal zero**
  - **'0'; # String zero char** **''; # Empty string**
  - **undef; # Undefined**
- **Logic Operators**
  - **$a = 0; $b = 45; # More than 1 statement per line possible**
  - **print($a and $b++); # prints 0 ***
  - **$a = 22;**
  - **print($a and $b++); # prints 45 ***
  - **print $b; # prints 46/ $b++ only eval when $a true**

Understand Perl

- **Logic Operators**
  - **or = Logical OR**
  - **|| = Logical OR**
  - **and = Logical AND**
  - **&& = Logical AND**
  - **not = Logical NOT**
  - **! = Logical NOT**
  - **| = Bitwise O**
  - **& = Bitwise AND**
  - **~ = Bitwise NOT**

Understand Perl

- **Logic Operators**
  - **print 6 & 5;   # prints 4, 0b0110 & 0b0101 = 0b0100**
  - **print 6 | 5;   # prints 7, 0b0110 | 0b0101 = 0b0111**
  - **print ! 0;   # prints 1**
  - **print ! 5;   # prints nothing (that is undef or false)**
  - **print ~5;   # prints 4294967290, same as:**
  - **# 0b11111111111111111111111111111010**

Understand Perl

- **Arrays**
  - **@components = ('X_LUT4', 'X_AND2', 'X_BUFGMUX', 'X_BUF_PP', 'X_FF');**
  - **# or use qw (Quoted Words), saves typing commas or quotes, gives the same result**
  - **@components = qw'X_LUT4 X_AND2 X_BUFGMUX X_BUF_PP X_FF';**
  - **push(@components, 'X_MUX2'); # Push item onto the top**
  - **print $components[0];   # Prints element 0**
  - **print "@components\n";   # Prints separated by spaces**
  - **print @components   ;**

Understand Perl

- **Sort**
  - **(sort @array) ## sort alphabetically, with uppercase first**
  - **(sort {$a <=> $b} @array) ## sort numerically**
  - **(sort {$b cmp $a} @array) ## sort reverse alphabetically**

Understand Perl

- **Command Line Arguments**
  - **$script_filename = $ARGV[0];**
  - **$report_filename = $ARGV[1];**
  - **print "   Processing $script_filename\n";**
  - **print "   Writing report to $report_filename\n";**
  - **print "   ARGV contains '@ARGV'\n";**

Understand Perl

- **Conditions**
  - **if($ff_count == 1) {**

```
    print "There is 1 flip flop\n";          # true
} else {
    print "There are $ff_count flip flops\n";       #false
}
```

Understand Perl

- **While**
  - **while($count < 100) {**

```
while( $count < 100 ) {
  $count++;          # Perl assumes $count == 0 the first time
  print "$count\n";
}
```

Understand Perl

- **Foreach**
  - **foreach $course ('perl', 'python', 'c', 'bash') {**

```
foreach $course ( 'perl', 'python', 'c', 'bash' ) {
  print "There is a $course training course\n";
}
```

  - **foreach $component (@components) {**

```
foreach $component ( @components ) {
  print "Component is $component\n";
}
```

Understand Perl

- **Files**
  - **open(FILE1, 'file1.txt'); # read mode, default**
  - **open(FILE1, '>file1.txt'); # write mode**
  - **print FILE1 "The first line to file1.txt\n";**
  - **print FILE1 "The final line to file1.txt\n";**
  - **close(FILE1);   # Don't have to explicitly close a file**
  - **$first_line = <FILE2>; # reading first line file 2 into first_line**
  - **while($line = <FILE2>) {**
print $line;   # Rea/ print rest of lines from file2.txt.
}

Understand Perl

- STDOUT/STDIN
  - print STDOUT "This goes to the standard output\n";
  - print   "So does this\n";
  - $standard_input = <STDIN>;   # Read line from standard input.
  - chomp($standard_input);   # Remove the trailing newline character

Understand Perl

- **Pattern matching**
  - **$string = "Novice to Expert in a 3 day Perl course.\n";**

```
print $string;
if( $string =~ m/Expert/ ) {
  # successful match returns 1 so this statement is executed
  print "This string contains the substring 'Expert'\n";
}
```

  - **m stands for match, forward slashes are used to /delimit/ regular expressions, =~ tells the m operator which string to search, the m is optional when // are used**

Understand Perl

- **Regular Expressions**
  - **use English;**

```
$string = "Novice to Expert in a 3 day Perl course.\n";
if( $string =~ /\w+/ ) {
  # \w+ matches alphanumeric characters in a row
  print "Matched: $MATCH\n";  # Matched: Novice
}
```

Understand Perl

- **Regular Expressions**
  - **use English;**

```
$string = "Novice to Expert in a 3 day Perl course.\n";
if( $string =~ /Perl\s+\w+/ ) {
  #            ^^^^          matches Perl
  #                ^^^       matches white space characters
  #                             (including space, tab and newline)
  #                   ^^^    matches alphanumeric characters
  print "Matched: $MATCH\n";  # Matched: Perl course
}
```

Understand Perl

- **Socket Listener**
  - **use IO::Socket;**
  - **my $sock = new IO::Socket::INET (**
  - **LocalHost => 'myhost',**
  - **LocalPort => '7070',**
  - **Proto => 'tcp',**
  - **Listen => 1,**
  - **Reuse => 1,**
  - **);**
  - **die "Could not create socket: $!\n" unless $sock;**

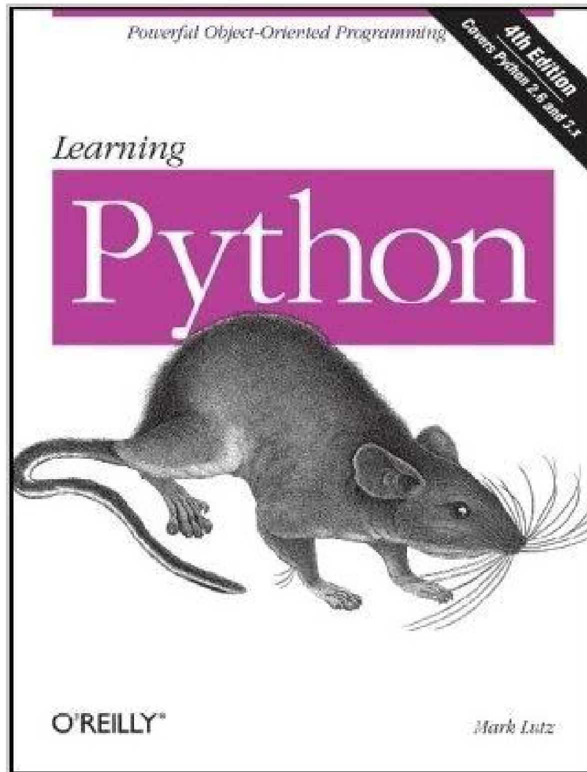
Understand Perl

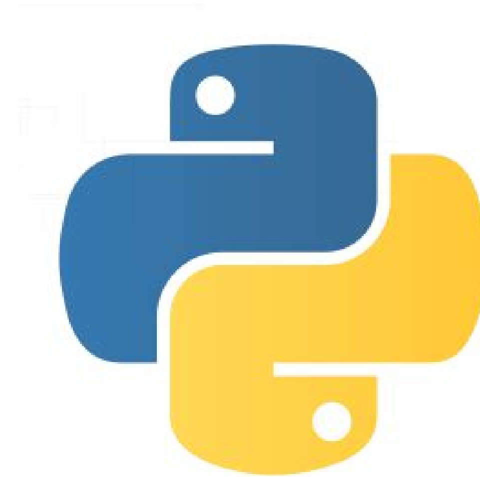
- **Socket Listener**
  - **my $new_sock = $sock->accept();**
    - **while(<$new_sock>) {**
    - **print $_;**
    - **}**
    - **close($sock);**

Understand Perl

- **Socket Caller**
  - **use IO::Socket;**
  - **my $sock = new IO::Socket::INET (**
  - **PeerAddr => 'host',**
  - **PeerPort => '7070',**
  - **Proto => 'tcp',**
  - **);**
  - **die "Could not create socket: $!\n" unless $sock;**
  - **print $sock "Hello there!\n";**
  - **close($sock);**

Understand Python





Understand Python

- Python
  - print 'hello world'
- File
  - cat > hello.py
  - print 'hello world”
  - ^D
  - python hello.py
- AAAA... ;)
  - print 'A'*30
- Labels and pointers
  - label1 = 'Dilbert'
  - label2 = label1
  - label1 = 'Dogbert'
  - label2

Understand Python

- Strings
  - string1 = 'Dilbert'
  - string2 = 'Dogbert'
  - string1 + string2
  - string2[2:4]
  - string1[0]
- Strings
  - len(string2)
  - string2[0:]
  - string1[-5:]
  - string2.find('og')
  - string2.replace('og','ago')

Understand Python

- **Numbers**
  - **n1=5**
  - **n2=3**
  - **n2 * n1**
  - **n1 ** n2 # power of**
  - **5 / 3, 5 % 3 # modulus**
  - **n3 = 1**
  - **n3 << 3**
- **Numbers**
  - **s1 = 'abc'**
  - **n1 = 12**
  - **s1 + n1**
  - **s1 + str(n1)**
  - **s1.replace('c'str(n1))**
  - **s1*n1**
  - **x1 = 5**
  - **x1 = n1 ** 2**

Understand Python

- Lists
  - mylist = [1,2,3]
  - len(mylist)
  - mylist*4
  - 1 in mylist
  - mylist[1:]
  - biglist = [['Dilbert', 'Dogbert'], ['Wally','Alice']]
  - biglist[1][0]

- Lists
  - biglist[0][1]
  - biglist[1] = 'Ratbert'
  - stacklist = biglist[0]
  - stacklist = stacklist + ['The Boss']
  - stacklist.pop()
  - stacklist.extended(['lol'])
  - stacklist.reverse()

Understand Python

- Dictionaries
  - d = { 'hero' : 'Dilbert' }
  - d['hero']
  - 'hero' in d
  - 'Dilbert' in d
  - d.keys()
  - d.values()
  - d['hero'] = 'Dogbert'
- Dictionaries
  - d['buddy'] = 'Wally'
  - d['pets'] = 2
  - d

Understand Python

- **Files**

```
cat targets
  RPC-DCOM  10.10.20.1,10.10.20.4
  SQL-SA              10.10.20.27,10.10.20.28
targets_file = open('targets','r')
lines = targets_file.readlines()
lines_dictionary = {}
for line in lines:
  one_line = line.split()
  line_key = one_line[0]
```

Understand Python

- **Files**

```
  lines_dictionary[line_key] = line_value

for key in lines_dictionary.keys():
  target_string = lines_dictionary[key]
  target_list = targets_string.split(',')
  targets_number = len(targets_list)
  filename = key + '_' + str(targets_number) + '_targets'
  vuln_file = open(filename,'w')
```

Understand Python

- **Files**

```
for vuln_target in targets_list:
  vuln_file.write(vuln_target + '\n')
vuln_file.close()
```

- **For**
  - **for <iterator-value> in <list to iterate over>:**
    - **ends with ':', always tab-in**
    - **whitespaces and tabs are used as code markings!**

Understand Python

- If
  - if foo > 3:
    - print 'Foo greater than 3'
  - elif foo == 3:
    - print 'foo equals 3'
  - else
    - print 'foo not greater than or equal to 3'
- While
  - while foo < 10:
    - foo = foo + bar

Understand Python

- **Sockets**

```
nc -l -p 4141

import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('localhost', 4141))
s.send('hello', 'world')
data = s.recv(1024)
s.close()
print 'Received', 'data'
```

Understand Shellscripting

- **Bash = Bourne Again Shell**
- **GNU GPL**

Understand Shellscripting

- Redirection
  - ls -l > ls-l.txt # standard out
  - grep da * 2> grep-errors.txt # error redir
  - grep da * 1>&2 # all standard error
  - grep * 2>&1 # all standard out
  - rm -f $(find / -name core) &> /dev/null
- Pipes
  - ls -l | sed -e "s/[aeio]/u/g"
  - ls -l | grep "\.txt$"

Understand Shellscripting

- Variables
  - STR="Hello World!"
    - echo $STR
  - $(date +%Y%m%d)
- Conditionals
  - if ["foo" = "foo"]; then

```
if [ "foo" = "foo" ]; then
        echo expression evaluated as true
fi
```

Understand Shellscripting

- **Conditionals**

```
if [ "foo" = "foo" ]; then
        echo expression evaluated as true
else
        echo expression evaluated as false
fi
```

Understand Shellscripting

- Conditionals
  - ```
    T1="foo"
    T2="bar"
    if [ "$T1" = "$T2" ]; then
            echo expression evaluated as true
    else
            echo expression evaluated as false
    fi
    ```
    ```

Understand Shellscripting

- For
  - for i in $(ls); do

```
for i in $( ls ); do
        echo item: $i
 done
```

  - for i in `seq 1 10`;

```
for i in `seq 1 10`;
do
    echo $i
done
```

Understand Shellscripting

- While
  - COUNTER=0

```
while [  $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

Understand Shellscripting

- **Until**

```
COUNTER=20
until [  $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

Understand Shellscripting

- Functions
  - function quit {

```
function quit {
  exit
}
function hello {
  echo Hello!
}
hello
quit
```

Understand Shellscripting

- **User Interface**
  - **OPTIONS="Hello Quit"**

```
OPTIONS="Hello Quit"
select opt in $OPTIONS; do
if [ "$opt" = "Quit" ]; then
      echo done
      exit
elif [ "$opt" = "Hello" ]; then
      echo Hello World
else
       echo bad option
fi
done
```

Understand Shellscripting

- **User Input**
  - **echo Please, enter your name**
    read NAME
    echo "Hi $NAME!"

  - **echo Please, enter your firstname and lastname**
    read FN LN
    echo "Hi! $LN, $FN !"

Understand Shellscripting

- **Arithmetic**
  - **echo 1 + 1**
  - **echo $((1+1))**
  - **echo $[1+1]**
  - **echo $[3/4]**
  - **echo 3/4|bc -l**

- **Arithmetic**
  - **-lt (<)**
  - **-gt (>)**
  - **-le (<=)**
  - **-ge (>=)**
  - **-eq (==)**
  - **-ne (!=)**

Understand Shellscripting

- **Strings**
  - **s1 = s2** **# matches**
  - **s1 != s2** **# no match**
  - **s1 < s2** **# lower than**
  - **s1 > s2** **# greater than**
  - **-n s1** **# not 0**
  - **-z s1** **# is 0**

Understand Shellscripting

- function Usage(){
- echo " Help"
- echo "usage: $0 IP"
- echo "example: $0 192.168.0.10"
- }

- # parameter checking
- if [$# -ne 1]
- then
- Usage
- exit 1
- fi

Understand Shellscripting

- **signal_handler()**
- **{**
- **sync**
- **echo "   actual test aborted..."**
- **}**

- **# catch ctrl+c signal**
- **trap signal_handler SIGINT**

More Tools

- **A lot of GNU Utils**
- **Unix/Linux and Win32**
- **Small, scriptable**

More Tools

- grep
  - cat *.txt | grep "search string"
  - grep -r "search string" /tmp
  - grep "string" file
  - grep -v "string" file
- sort
  - sort /tmp/dummy
- unique
  - unique /tmp/dummy

More Tools

- **sed**
  - **sed 's/to_be_replaced/replaced/g' /tmp/dummy**
  - **sed 12, 18d /tmp/dummy**
  - **sed -i -e 's/HOSTNAME.*/HOSTNAME="mybox"/' conf.d/hostname**
  - **sed -e 's/.*Request: //' -e 's#[/:].**##' file**
- **wc**
  - **wc --words --lines --bytes /tmp/dummy**

More Tools

- **awk**
  - **awk '/test/ {print}' /tmp/dummy**
  - **awk '/test/ {i=i+1} END {print i}' /tmp/dummy**
  - **awk -F: '{print $2}' john.pot > pw.txt**
  - **awk -F: '{print $1}' /etc/passwd > user.txt**
  - **awk '{print $3}' file**
  - **awk -F, '{printf"%s,%s,%s\n", $2,$1,$3}'**
  - **awk -F, -v OFS=, '{tmp=$1; $1=$2; $2=tmp; print}'**

More Tools

- log=`date +%Y%m%d_%k%M`_$1.log
- command 2>&1 | tee -a $log
- script FILENAME.txt
- ifconfig eth0 > file.txt
- route > file.txt
- tcpdump -i ethX -n -vvv -s0 -XX -w FILE host X.X.X.X
- tcpdump -i ethX -n -vvv -s0 -XX -w FILE net X.X.X.X/24
- tcpdump -i ethX -n -vvv host TARGETIP | grep ">"
- echo -e "GET HTTP/1.0\n\n" | nc -vv TARGETIP 80
- echo -e "GET HTTP/1.0\n\n" | openssl s_client -quiet -connect TARGETIP:443
- echo -e "OPTIONS * HTTP/1.0\n\n\n" | nc TARGETIP 80
- echo -e "HEAD / HTTP/1.0\n\n\n" | nc TARGETIP 80

Coding Online

- http://www.ethicalhacker.net/content/view/82/2/
- http://www.comp.nus.edu.sg/~hugh/TeachingStuff/cs1101c.pdf
- http://www.le.ac.uk/cc/tutorials/c/
- http://computer.howstuffworks.com/c.htm
- http://www.clifford.at/papers/2005/buffer/phrack/p49-14.txt
- http://computer.howstuffworks.com/c23.htm
- http://www.groar.org/expl/beginner/buffer1.txt
- http://www.rdrop.com/~cary/html/endian_faq.html
- http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html
- http://home.si.rr.com/mstoneman/pub/docs/Processors%20History.rtf
- http://webster.cs.ucr.edu/
- http://www.ccntech.com/code/x86asm.txt
- http://www.gnu.org/software/gdb/documentation/
- http://www.perl.org/docs.html
- http://docs.python.org/index.html
- http://www.gnu.org/software/bash/manual/bashref.html
- http://www.indianz.ch/tools/doc/commands.txt

Besten Dank...

… für Ihre Aufmerksamkeit!

Wem darf ich eine Frage beantworten? ;-)

IndianZ
www.indianz.ch