HACKTHEBOX

INTRO TO ASSEMBLY LANGUAGE

CHEAT SHEET

Registers

Description	64-bit Register (8-bytes)	8-bit Register (1-bytes)
Data/Arguments Registers		
Syscall Number/Return value	`rax`	`al`
Callee Saved	`rbx`	`bl`
1st arg	`rdi`	`dil`
2nd arg	`rsi`	`sil`
3rd arg	`rdx`	`dl`
4th arg - Loop Counter	`rcx`	`cl`
5th arg	`r8`	`r8b`
6th arg	`r9`	`r9b`
Pointer Registers		
Base Stack Pointer	`rbp`	`bpl`
Current/Top Stack Pointer	`rsp`	`spl`
Instruction Pointer 'call only'	`rip`	`ipl`

Assembly and Disassembly

Command	Description
`nasm -f elf64 helloWorld.s`	Assemble code
`ld -o helloWorld helloWorld.o`	Link code
`ld -o fib fib.o -lc --dynamic-linker /lib64/ld-linux-x86-64.so.2`	Link code with libc functions
`objdump -M intel -d helloWorld`	Disassemble `.text` section
`objdump -M intel --no-show-raw-insn --no-addresses -d helloWorld`	Show binary assembly code
`objdump -sj .data helloWorld`	Disassemble `.data` section

GDB

Command	Description
`gdb -q ./helloWorld`	Open binary in gdb
`info functions`	View binary functions
`info variables`	View binary variables
`registers`	View registers
`disas _start`	Disassemble label/function
`b _start`	Break label/function
`b *0x401000`	Break address
`r`	Run the binary

Command	Description
`x/4xg $rip`	Examine register "x/ count-format-size $register"
`si`	Step to the next instruction
`s`	Step to the next line of code
`ni`	Step to the next function
`c`	Continue to the next break point
`patch string 0x402000 "Patched!\\x0a"`	Patch address value
`set $rdx=0x9`	Set register value

Assembly Instructions

Instruction	Description	Example
Data Movement		
`mov`	Move data or load immediate data	`mov rax, 1` -> `rax = 1`
`lea`	Load an address pointing to the value	`lea rax, [rsp+5]` -> `rax = rsp+5`
`xchg`	Swap data between two registers or addresses	`xchg rax, rbx` -> `rax = rbx, rbx = rax`
Unary Arithmetic Instructions		
`inc`	Increment by 1	`inc rax` -> `rax++` or `rax += 1` -> `rax = 2`
`dec`	Decrement by 1	`dec rax` -> `rax--` or `rax -= 1` -> `rax = 0`

Instruction	Description	Example
Binary Arithmetic Instructions		
`add`	Add both operands	`add rax, rbx` -> `rax = 1 + 1` -> `2`
`sub`	Subtract Source from Destination (*i.e* ***`rax = rax - rbx`***)	`sub rax, rbx` -> `rax = 1 - 1` -> `0`
`imul`	Multiply both operands	`imul rax, rbx` -> `rax = 1 * 1` -> `1`
Bitwise Arithmetic Instructions		
`not`	Bitwise NOT (*invert all bits, 0->1 and 1->0*)	`not rax` -> `NOT 00000001` -> `11111110`
`and`	Bitwise AND (*if both bits are 1 -> 1, if bits are different -> 0*)	`and rax, rbx` -> `00000001 AND 00000010` -> `00000000`
`or`	Bitwise OR (*if either bit is 1 -> 1, if both are 0 -> 0*)	`or rax, rbx` -> `00000001 OR 00000010` -> `00000011`
`xor`	Bitwise XOR (*if bits are the same -> 0, if bits are different -> 1*)	`xor rax, rbx` -> `00000001 XOR 00000010` -> `00000011`
Loops		
`mov rcx, x`	Sets loop (`rcx`) counter to `x`	`mov rcx, 3`
`loop`	Jumps back to the start of `loop` until counter reaches `0`	`loop exampleLoop`
Branching		

HTB ACADEMY CHEATSHEET

Instruction	Description	Example
`jmp`	Jumps to specified label, address, or location	`jmp loop`
`jz`	Destination **equal to Zero**	`D = 0`
`jnz`	Destination **Not equal to Zero**	`D != 0`
`js`	Destination **is Negative**	`D < 0`
`jns`	Destination **is Not Negative** (i.e. 0 or positive)	`D >= 0`
`jg`	Destination **Greater than** Source	`D > S`
`jge`	Destination **Greater than or Equal** Source	`D >= S`
`jl`	Destination **Less than** Source	`D < S`
`jle`	Destination **Less than or Equal** Source	`D <= S`
`cmp`	Sets `RFLAGS` by subtracting second operand from first operand (*i.e. first - second*)	`cmp rax, rbx` -> `rax - rbx`
Stack		
`push`	Copies the specified register/address to the top of the stack	`push rax`
`pop`	Moves the item at the top of the stack to the specified register/address	`pop rax`
Functions		
`call`	push the next instruction pointer `rip` to the stack, then jumps to the specified procedure	`call printMessage`
`ret`	pop the address at `rsp` into `rip`, then jump to it	`ret`

HTB ACADEMY CHEATSHEET

Functions

Command	Description
`cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep write`	Locate `write` syscall number
`man -s 2 write`	`write` syscall man page
`man -s 3 printf`	`printf` libc man page

Syscall Calling Convention

1. Save registers to stack
2. Set its syscall number in `rax`
3. Set its arguments in the registers
4. Use the `syscall` assembly instruction to call it

Function Calling Convention

1. `Save Registers` on the stack (***`Caller Saved`***)
2. Pass `Function Arguments` (*like syscalls*)
3. Fix `Stack Alignment`
4. Get Function's `Return Value` (*in* ***`rax`***)

Shellcoding

Command	Description
`pwn asm 'push rax' -c 'amd64'`	Instruction to shellcode
`pwn disasm '50' -c 'amd64'`	Shellcode to instructions
`python3 shellcoder.py helloworld`	Extract binary shellcode
`python3 loader.py '4831..0f05`	Run shellcode
`python assembler.py '4831..0f05`	Assemble shellcode into binary

Shellcraft

Command	Description
`pwn shellcraft -l 'amd64.linux'`	List available syscalls
`pwn shellcraft amd64.linux.sh`	Generate syscalls shellcode
`pwn shellcraft amd64.linux.sh -r`	Run syscalls shellcode

Msfvenom

`msfvenom -l payloads | grep 'linux/x64'`	List available syscalls
`msfvenom -p 'linux/x64/exec' CMD='sh' -a 'x64' --platform 'linux' -f 'hex'`	Generate syscalls shellcode
`msfvenom -p 'linux/x64/exec' CMD='sh' -a 'x64' --platform 'linux' -f 'hex' -e 'x64/xor'`	Generate encoded syscalls shellcode

Shellcoding Requirements

1. Does not contain variables
2. Does not refer to direct memory addresses
3. Does not contain any NULL bytes `00`

HTB ACADEMY CHEATSHEET