CSLU IIUM

Reverse Engineering
0x251e

Table of contents:

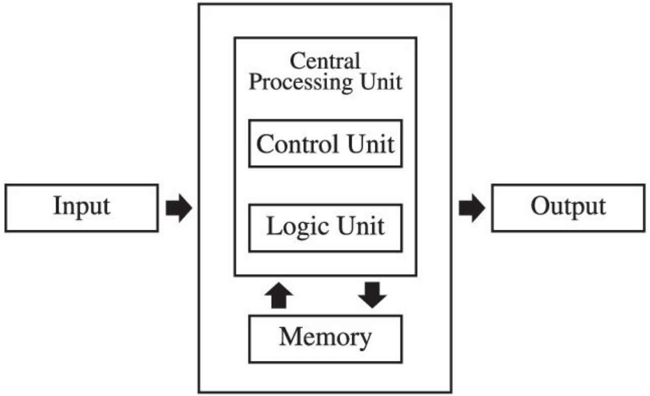
Theory:

Hands-on Labs: https://github.com/shreethaar/CSLU-IIUM-RE-HandsOnSession

Introduction to x86

In order to understand x86 architecture, we should familiar with Von Neumann architecture:

1. Control Unit
2. Arithmetic Logic Unit (ALU)
3. Registers
4. Memory
5. I/O devices



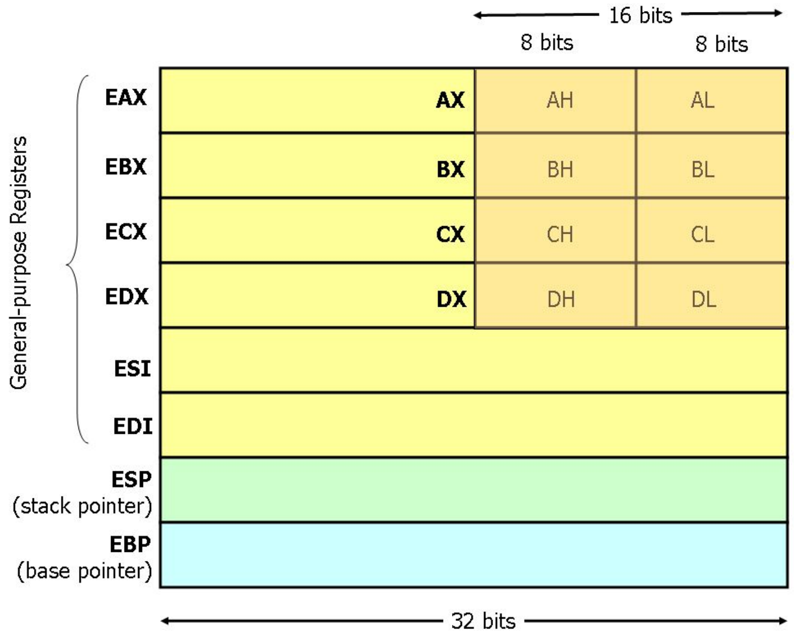
Introduction to x86

What is the Control Unit:

- Direct and coordinate execution of instruction in CPU

- Receives instructions from main memory

- In 32-bit systems, register that handles the function of control unit is Extended Instruction Pointer **(EIP)**



Introduction to x86

What is the ALU:

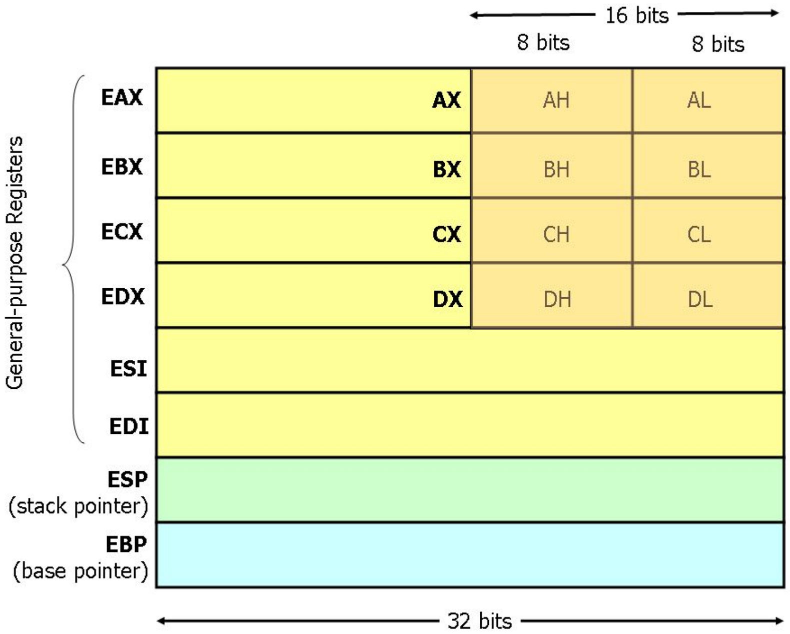
- perform all calculations and logic operations

- works under direction of Control Unit

- Executes arithmetic, logic and comparison

- Result of operation stored in registers or memory

- In the form of assembly instruction such as **XOR, ADD, CMP** and etc



Introduction to x86

What is registers:

- small, fast storage inside CPU

- used during instruction execution

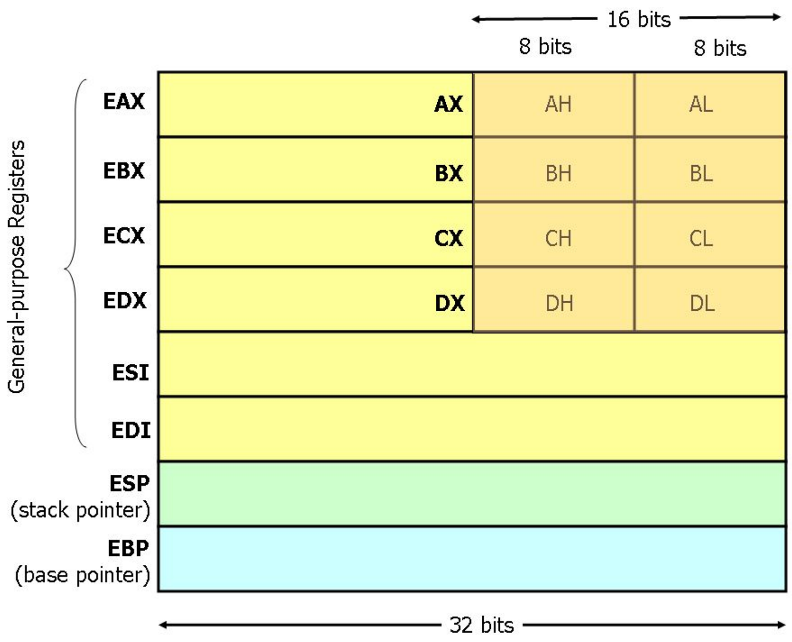
- provide quick access to frequently used values

Types of registers:

- Data register **(EAX, EBX, ECX and EDX)**

- Address register **(ESI and EDI)**

- Control register **(EIP, ESP and EBP)**



Introduction to x86

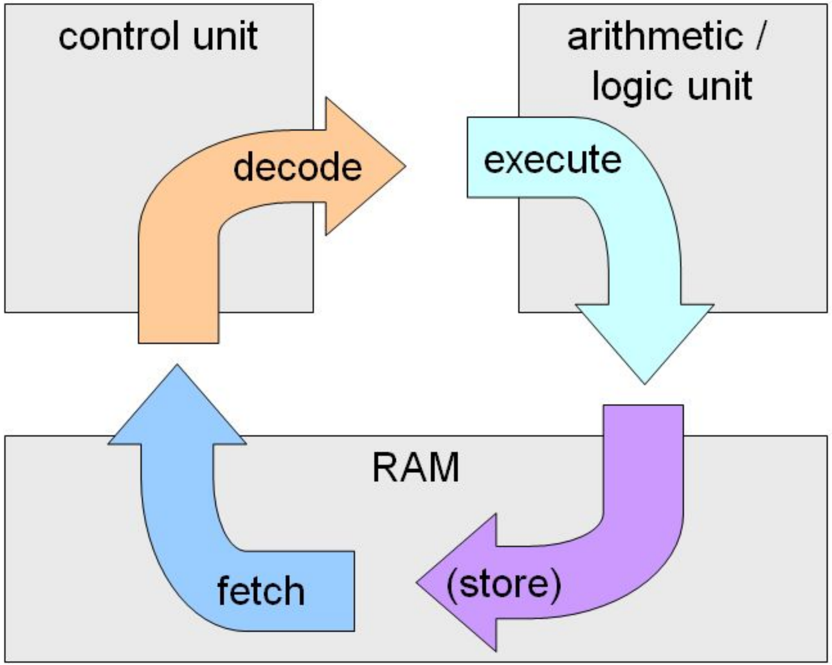
What is the memory ?

- workspace where programs and data are stored while running

- when a program starts, it is loaded from disk into memory

- CPU fetch instruction from memory using **Instruction Pointer**

- allow direct access to any address (random access memory)



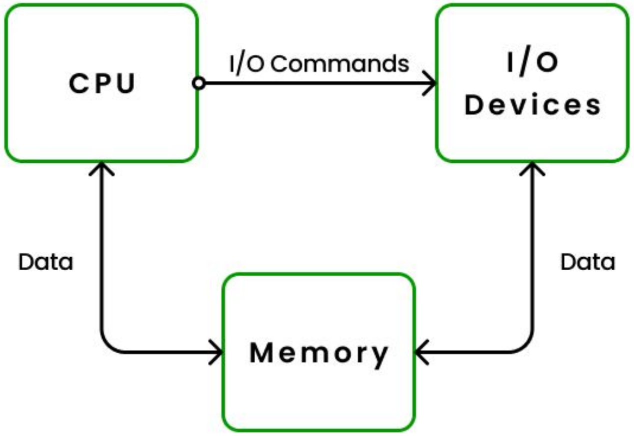
Introduction to x86

What is the I/O devices:

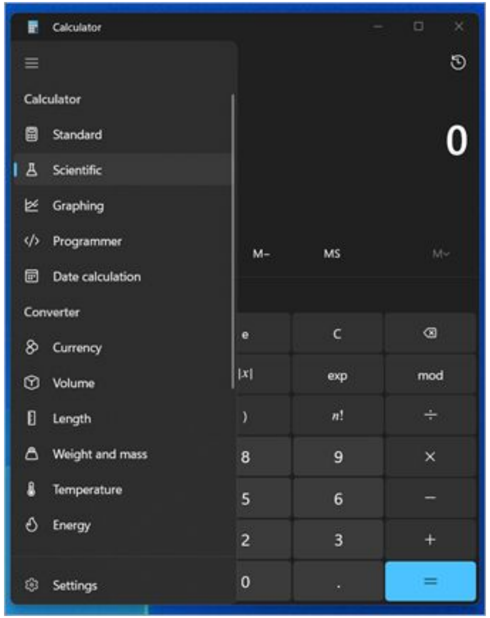
- to communicate with peripherals

- send input to CPU or receive output

- in Linux, programs interact with I/O device using system calls



Compilation & translation



So how does applications is made and understand by CPU?

Take example of this calculator applications

1. Programmers with it in a high level language
2. Compiler translates it into Assembly code
3. CPU execute it in binary (1s and 0s)

A program you see on screen is the final product of many translation steps.
It is from human-readable code to CPU-executable machine code.

Compilation & translation

What if we are given only the application without a single clue what is the source code is

This is the process of reverse engineering needed

Key Takeway:
“Compilation is a one-way translation — **Reverse Engineering** is learning how to read it backward.”

“hang pi taruq gear R, astu gostan, ape yg susah” - 0x251e

C and NASM in Linux

1. C language is human-readable which computer can’t execute directly
2. When we compile C:
   - **Preprocessor** expands macros such as **#include**, **#define**
   - **Compiler** converts C -> assembly (.s)
   - **Assembler** turns assembly -> machine code (.o)
   - **Linker** joins **.o** + libraries -> executables (a.out)
3. NASM (Netwide Assembler), popular assembler for Linux

With C code, you compile with GCC compiler

With Assembly code, you compile with NASM with required to link the object files

Bridging C code to assembly

A basic software/program contains:

- variable definition (char, int, long, array)
- if/else conditions
- loops (while,for)
- calling functions

```
Enter two numbers :
1
2
Enter Choice
1 - Add
2 - Sub
3 - Mul
4 - Div
1
Result: 3
```

Questions:
Based on the application on the right side, what are the possible variables and conditions ?

Bridging C code to assembly

Two possible variables:

`num1` -> holding the first value

`num2` -> holding the second value

Conditions:

If 1: add
If 2: sub
If 3: mul
If 4: div

Questions:
how about more than 4 ???

```
Enter two numbers :
1
2
Enter Choice
1 - Add
2 - Sub
3 - Mul
4 - Div
1
Result: 3
```

Bridging C code to assembly

```c
#include <stdio.h>

int main() {
    int num1, num2, result;
    int choice;

    printf("Enter two numbers: \n");
    scanf("%d", &num1);
    scanf("%d", &num2);

    printf("Enter Choice\n");
    printf("1 - Add\n");
    printf("2 - Sub\n");
    scanf("%d", &choice);
```

```c
    if (choice == 1) {
        result = num1 + num2;
        printf("Result: %d\n", result);
    }

    else if (choice == 2) {
        result = num1 - num2;
    printf("Result: %d\n", result);
    }


    else {
        printf("Invalid choice\n");
        }
    return 0;
}
```

Bridging C code to assembly

Now we have to source code, to make it executable as a program, we need to compile it:

```
$ gcc -m32 calc.c -o calc
```

```
[trevorphilips@allSafe simple-calc]$ ls
calc.c
[trevorphilips@allSafe simple-calc]$ gcc -m32 calc.c -o calc
[trevorphilips@allSafe simple-calc]$ ls
calc  calc.c
[trevorphilips@allSafe simple-calc]$ file calc
calc: ELF 32-bit LSB pie executable, Intel i386, version 1 (SYSV), dynamically linked, interpreter /lib
/ld-linux.so.2, BuildID[sha1]=043757268cdaf14e104fcf6409b7936a8f77182a, for GNU/Linux 4.4.0, not stripp
ed
[trevorphilips@allSafe simple-calc]$
```

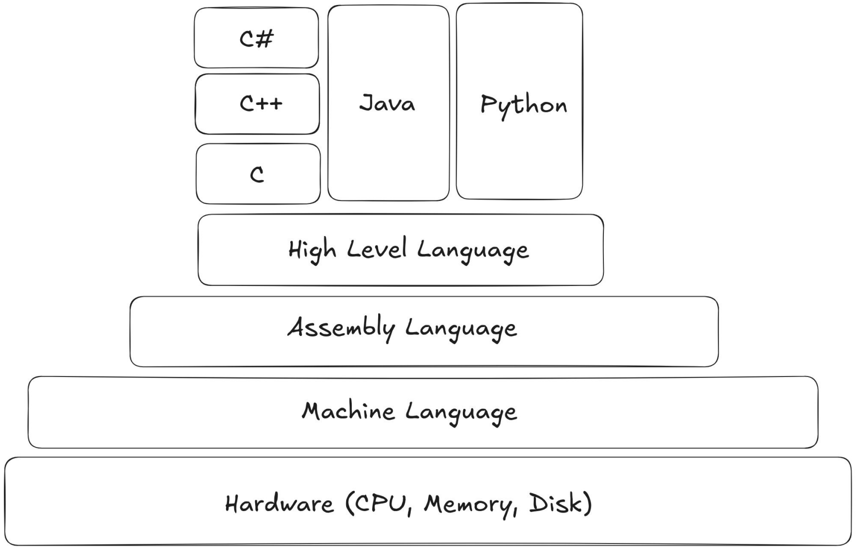
Bridging C code to assembly

When we run the calc, how does hardware process the code ?
Computers only understand binary (1 and 0)

Here is when **assembly** translates high level source code to **machine code** for hardware to process

To view the source code of the calc code:

$ gcc -m32 -S -masm=intel -O2 -fno-asynchronous-unwind-tables calc.c -o calc.s



Bridging C code to assembly

```
push ebp
mov  ebp, esp
push esi
push ebx
call __x86.get_pc_thunk.bx
add  ebx, OFFSET
FLAT:_GLOBAL_OFFSET_TABLE_
sub  esp, 40
mov  eax, DWORD PTR gs:20
mov  DWORD PTR -28[ebp], eax
```

This part of assembly is prologue:

EBP -> Base Pointer
ESP -> Stack Pointer
SUB -> Subtract 40 bytes from ESP

Purpose is to:
- Make space to store local variables
- Save caller address

Why **`sub esp, 40`**:
- num1 is an int, 4 bytes
- num2 is an int, 4 bytes
- choice is an int, 4 bytes
- the remaining is for stack operation later usage

Bridging C code to assembly

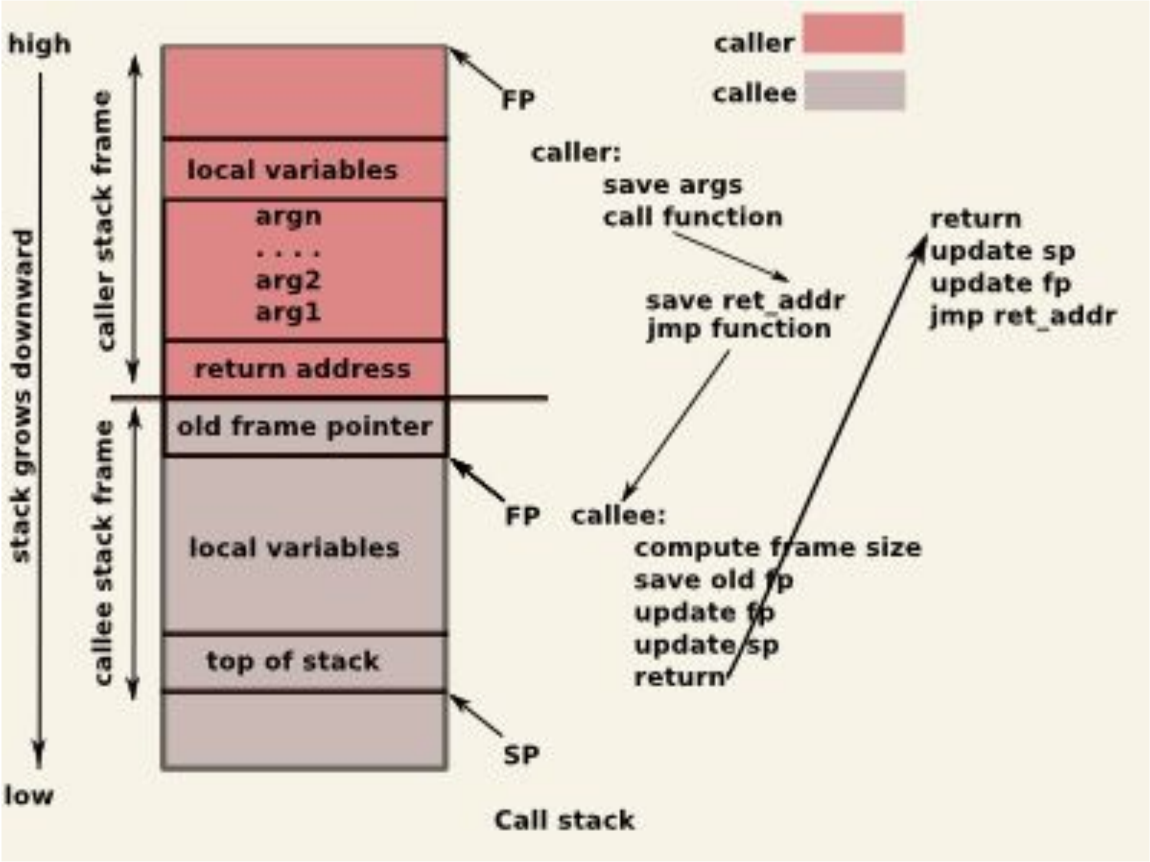
What is a stack ?

- a region of memory used for temporary storage

- grows downwards

- each function has its own stack frame

- local variables



Bridging C code to assembly

```
lea eax, .LC0@GOTOFF[ebx]
push eax
call puts@PLT
```

CALL -> to execute another function **puts**
[ebx] is referring to the address that is in EBX, then calls puts to print out string

```
lea eax, -40[ebp]
push eax
push esi
call __isoc23_scanf@PLT
```

[ebp-40] from stack where the address of num1 will be stored

CALL -> to execute C library function **scanf**

```
lea eax, -36[ebp]
push eax
push esi
call __isoc23_scanf@PLT
```

[ebp-36] from stack where the address of num2 will be stored

CALL -> to execute C library function **scanf**

Bridging C code to assembly

```
.L9:
    pushedx
    pushedx
    mov eax, DWORD PTR -36[ebp]
    add eax, DWORD PTR -40[ebp]
    jmp .L7
```

Add operations:
1. First number move into EAX
2. **Add** second number and store in EAX
3. Push EAX and then call printf

JMP .L7 is to print result

Bridging C code to assembly

```
.L7:
    pusheax
    lea eax, .LC5@GOTOFF[ebx]
    pusheax
    callprintf@PLT
    add esp, 16
    jmp .L3
```

Print result operations:
Result from operations stored in EAX push into stack

Next push is the format string, then we call printf

In C code, printf contain two arguments

printf(**"Result: %d\n"**, **result**);

Bridging C code to assembly

```
.L10:
    pusheax
    pusheax
    mov eax, DWORD PTR -40[ebp]
    sub eax, DWORD PTR -36[ebp]
```

Would someone like to give a try to explain this ?

Grab some free stickers

Bridging C code to assembly

```
call__isoc23_scanf@PLT
mov eax, DWORD PTR -32[ebp]
add esp, 16
cmp eax, 1
je  .L9
cmp eax, 2
je  .L10
sub esp, 12
lea eax, .LC6@GOTOFF[ebx]
pusheax
callputs@PLT
add esp, 16
```

Compare value for calculation operations

1. cmp eax 1 jumps to addition
2. cmp eax 2 jumps to subtraction

JE is jump equals

So what is LEA,PUSH and CALL puts does ?

C Code Review

```c
#include <stdio.h>

int main() {
    char ch;

    printf("Enter a character: ");
    scanf("%c", &ch);
    printf("\nCharacter: %c\n", ch);
    printf("ASCII value: %d\n", ch);
    printf("Hexadecimal: 0x%X\n", ch);
    if (ch >= 'A' && ch <= 'Z') {
        printf("Type: Uppercase letter\n");
    }
    else if (ch >= 'a' && ch <= 'z') {
        printf("Type: Lowercase letter\n");
    }
    else if (ch >= '0' && ch <= '9') {
        printf("Type: Digit\n");
    }
    else if (ch == ' ') {
        printf("Type: Space\n");
    }
    else {
        printf("Type: Special character\n");
    }

    return 0;
}
```

Here is a simple ascii checker:

1. Take a single character as input
2. Display input character (%c)
3. Display in ASCII value (%d)
4. Display in hexadecimal value (0%x%x)
5. Identify type of character input

To compile and debug this:

$ gcc -m32 -g ascii-check.c -o check

C Code Review

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001030  puts@plt
0x0000000000001040  __stack_chk_fail@plt
0x0000000000001050  printf@plt
0x0000000000001060  __isoc23_scanf@plt
0x0000000000001070  _start
0x0000000000001169  main
0x00000000000012ac  _fini
pwndbg>
```

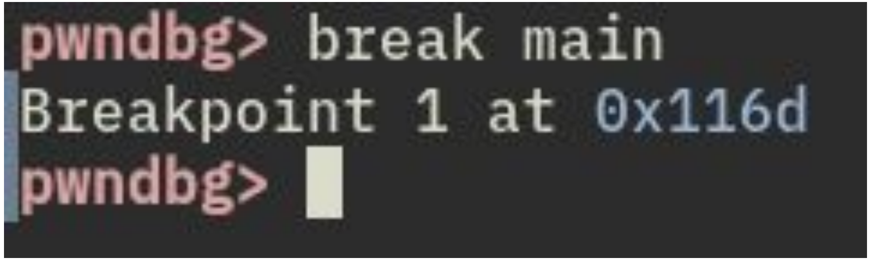
Run:

`$ pwndbg ./ascii-check`

Within gdb/pwndbg:
`$ info functions`

This will list the functions in the binary

C Code Review



$ break main

This set a breakpoint at main

Think breakpoint as a brake, it will pause the program until the address 0x116d

and next hit run:

$ run

C Code Review

```
Breakpoint 1, 0x000055555555516d in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
─────────────────────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────────────────────
 RAX  0x7ffff7e10e28 (environ) —▸ 0x7fffffffe2a8 —▸ 0x7fffffffe67d ◂— 'SHELL=/bin/bash'
 RBX  0
 RCX  0x555555557dd8 —▸ 0x555555555110 ◂— endbr64
 RDX  0x7fffffffe2a8 —▸ 0x7fffffffe67d ◂— 'SHELL=/bin/bash'
 RDI  1
 RSI  0x7fffffffe298 —▸ 0x7fffffffe631 ◂— '/home/trevorphilips/Desktop/cslu-uia-re/dist/source-code-review/ascii-check'
 R8   0x7ffff7e09680 —▸ 0x7ffff7e0afe0 ◂— 0
 R9   0x7ffff7e0afe0 ◂— 0
 R10  0x7fffffffdeb0 ◂— 0
 R11  0x203
 R12  0x7fffffffe298 —▸ 0x7fffffffe631 ◂— '/home/trevorphilips/Desktop/cslu-uia-re/dist/source-code-review/ascii-check'
 R13  1
 R14  0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2f0 —▸ 0x555555554000 ◂— 0x10102464c457f
 R15  0x555555557dd8 —▸ 0x555555555110 ◂— endbr64
 RBP  0x7fffffffe170 —▸ 0x7fffffffe210 —▸ 0x7fffffffe270 ◂— 0
 RSP  0x7fffffffe170 —▸ 0x7fffffffe210 —▸ 0x7fffffffe270 ◂— 0
 RIP  0x55555555516d (main+4) ◂— sub rsp, 0x10
──────────────────────────────────────────[ DISASM / x86-64 / set emulate on ]──────────────────────────────────
 ▶ 0x55555555516d <main+4>     sub    rsp, 0x10                      RSP => 0x7fffffffe160 (0x7fffffffe170 - 0x10)
   0x555555555171 <main+8>     mov    rax, qword ptr fs:[0x28]       RAX, [0x7ffff7f91768] => 0xd2f433784c9fd300
   0x55555555517a <main+17>    mov    qword ptr [rbp - 8], rax       [0x7fffffffe168] <= 0xd2f433784c9fd300
   0x55555555517e <main+21>    xor    eax, eax                       EAX => 0
   0x555555555180 <main+23>    lea    rax, [rip + 0xe7d]             RAX => 0x555555556004 ◂— 'Enter a character: '
   0x555555555187 <main+30>    mov    rdi, rax                       RDI => 0x555555556004 ◂— 'Enter a character: '
   0x55555555518a <main+33>    mov    eax, 0                         EAX => 0
   0x55555555518f <main+38>    call   printf@plt                    <printf@plt>

   0x555555555194 <main+43>    lea    rax, [rbp - 9]
   0x555555555198 <main+47>    lea    rdx, [rip + 0xe79]     RDX => 0x555555556018 ◂— 0x726168430a006325 /* '%c' */
   0x55555555519f <main+54>    mov    rsi, rax
───────────────────────────────────────────────────────[ STACK ]────────────────────────────────────────────────
00:0000│ rbp rsp 0x7fffffffe170 —▸ 0x7fffffffe210 —▸ 0x7fffffffe270 ◂— 0
01:0008│+008     0x7fffffffe178 —▸ 0x7ffff7c27675 ◂— mov edi, eax
02:0010│+010     0x7fffffffe180 —▸ 0x7ffff7fc2000 ◂— 0x3010102464c457f
03:0018│+018     0x7fffffffe188 —▸ 0x7fffffffe298 —▸ 0x7fffffffe631 ◂— '/home/trevorphilips/Desktop/cslu-uia-re/dist/source-code-review/ascii-check'
04:0020│+020     0x7fffffffe190 ◂— 0x1ffffe1d0
05:0028│+028     0x7fffffffe198 —▸ 0x555555555169 (main) ◂— push rbp
06:0030│+030     0x7fffffffe1a0 ◂— 0
07:0038│+038     0x7fffffffe1a8 ◂— 0x3b9dc053334ab1e4
─────────────────────────────────────────────────────[ BACKTRACE ]──────────────────────────────────────────────
 ▶ 0   0x55555555516d main+4
   1   0x7ffff7c27675 None
   2   0x7ffff7c27729 __libc_start_main+137
   3   0x555555555095 _start+37
```

C Code Review

```
Breakpoint 1, 0x000055555555516d in main ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
```

This shows the breakpoint that we set

1. Breakpoint 1 means the first breakpoint
2. Debugger is paused at the address **0x000055555555516d**

C Code Review

```
──────────────────────────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──
EAX  0xf7f952d4 (environ) —▸ 0xffffd44c —▸ 0xffffd67d ◂— 'SHELL=/bin/bash'
EBX  0x56558ff4 (_GLOBAL_OFFSET_TABLE_) ◂— 0x3eec
ECX  0xffffd390 ◂— 1
EDX  0xffffd3b0 —▸ 0xf7f90e0c ◂— 0x22cd2c
EDI  0x56558ee8 —▸ 0x56556140 ◂— endbr32
ESI  0
EBP  0xffffd378 —▸ 0xf7ffcca0 (_rtld_global_ro) ◂— 0
ESP  0xffffd360 —▸ 0xf7fbb380 —▸ 0xf7d64000 ◂— 0x464c457f
EIP  0x565561ba (main+29) ◂— mov eax, dword ptr gs:[0x14]
───────────────────────────────────────────────[ DISASM / i386 / set emulate on ]──
```

This is the registers section, always pay close attention the value contains

Sometimes it contains address
Sometimes it contains values

Depends on the contexts, which we need to understand assembly first

C Code Review

```
──────────────────────────────────────[ STACK ]──────────────
00:0000│ esp 0xffffd360 —▸ 0xf7fbb380 —▸ 0xf7d64000 ◂— 0x464c457f
01:0004│-014 0xffffd364 ◂— 0
... ↓        2 skipped
04:0010│-008 0xffffd370 —▸ 0xffffd390 ◂— 1
05:0014│-004 0xffffd374 —▸ 0xf7f90e0c ◂— 0x22cd2c
06:0018│ ebp 0xffffd378 —▸ 0xf7ffcca0 (_rtld_global_ro) ◂— 0
07:001c│+004 0xffffd37c —▸ 0xf7d86535 ◂— add esp, 0x10
──────────────────────────────────────[ BACKTRACE ]──────────
```

Stack frame grows downwards

Everytime when we see **PUSH** and **POP**,
The stack **INCREASE** and **DECREASE** in address value

Hands-on Session

Lets debug and try some crackme

No more slides after this, promised maximum slides is 30

Cheatsheet for gdb:
https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf