**CPU Registers**

Registers are the essential components of a CPU. Almost all registers offer a small amount of storage space where data can be temporarily stored. However, some of them have a particular function.

These registers will be divided into General registers, Control registers, and Segment registers. The most critical registers we need are the General registers. In these, there are further subdivisions into Data registers, Pointer registers, and Index registers.

**Data registers**

| **32-bit Register** | **64-bit Register** | **Description** |
| --- | --- | --- |
| EAX | RAX | Accumulator is used in input/output and for arithmetic operations |
| EBX | RBX | Base is used in indexed addressing |
| ECX | RCX | Counter is used to rotate instructions and count loops |
| EDX | RDX | Data is used for I/O and in arithmetic operations for multiply and divide operations involving large values |

**Pointer registers**

| **32-bit Register** | **64-bit Register** | **Description** |
| --- | --- | --- |
| EIP | RIP | Instruction Pointer stores the offset address of the next instruction to be executed |
| ESP | RSP | Stack Pointer points to the top of the stack |
| EBP | RBP | Base Pointer is also known as Stack Base Pointer or Frame Pointer thats points to the base of the stack |

**Stack Frames**

Since the stack starts with a high address and grows down to low memory addresses as values are added, the Base Pointer points to the beginning (base) of the stack in contrast to the Stack Pointer, which points to the top of the stack.

As the stack grows, it is logically divided into regions called Stack Frames, which allocate the required memory in the stack for the corresponding function. A stack frame defines a frame of data with the beginning (EBP) and the end (ESP) that is pushed onto the stack when a function is called.

Since the stack memory is built on a Last-In-First-Out (LIFO) data structure, the first step is to store the previous EBP position on the stack, which can be restored after the function completes. If we now look at the bowfunc function, it looks like following in GDB:

Pointer registers

(gdb) disas bowfunc

Dump of assembler code for function bowfunc:

0x0000054d <+0>: push ebp # <---- 1. Stores previous EBP

0x0000054e <+1>: mov ebp,esp

0x00000550 <+3>: push ebx

0x00000551 <+4>: sub esp,0x404

<...SNIP...>

0x00000580 <+51>: leave

0x00000581 <+52>: ret

The EBP in the stack frame is set first when a function is called and contains the EBP of the previous stack frame. Next, the value of the ESP is copied to the EBP, creating a new stack frame.

Pointer registers

(gdb) disas bowfunc

Dump of assembler code for function bowfunc:

0x0000054d <+0>: push ebp # <---- 1. Stores previous EBP

0x0000054e <+1>: mov ebp,esp # <---- 2. Creates new Stack Frame

0x00000550 <+3>: push ebx

0x00000551 <+4>: sub esp,0x404

<...SNIP...>

0x00000580 <+51>: leave

0x00000581 <+52>: ret

Then some space is created in the stack, moving the ESP to the top for the operations and variables needed and processed.

**Prologue**

Prologue

(gdb) disas bowfunc

Dump of assembler code for function bowfunc:

0x0000054d <+0>: push ebp # <---- 1. Stores previous EBP

0x0000054e <+1>: mov ebp,esp # <---- 2. Creates new Stack Frame

0x00000550 <+3>: push ebx

0x00000551 <+4>: sub esp,0x404 # <---- 3. Moves ESP to the top

<...SNIP...>

0x00000580 <+51>: leave

0x00000581 <+52>: ret

These three instructions represent the so-called Prologue.

For getting out of the stack frame, the opposite is done, the Epilogue. During the epilogue, the ESP is replaced by the current EBP, and its value is reset to the value it had before in the prologue. The epilogue is relatively short, and apart from other possibilities to perform it, in our example, it is performed with two instructions:

**Epilogue**

Epilogue

(gdb) disas bowfunc

Dump of assembler code for function bowfunc:

0x0000054d <+0>: push ebp

0x0000054e <+1>: mov ebp,esp

0x00000550 <+3>: push ebx

0x00000551 <+4>: sub esp,0x404

<...SNIP...>

0x00000580 <+51>: leave # <----------------------

0x00000581 <+52>: ret # <--- Leave stack frame

**Index registers**

| **Register 32-bit** | **Register 64-bit** | **Description** |
| --- | --- | --- |
| ESI | RSI | Source Index is used as a pointer from a source for string operations |
| EDI | RDI | Destination is used as a pointer to a destination for string operations |

Another important point concerning the representation of the assembler is the naming of the registers. This depends on the format in which the binary was compiled. We have used GCC to compile the bow.c code in 32-bit format. Now let's compile the same code into a 64-bit format.

**Compile in 64-bit Format**

Compile in 64-bit Format

student@nix-bow:~$ gcc bow.c -o bow64 -fno-stack-protector -z execstack -m64

student@nix-bow:~$ file bow64 | tr "," "\n"

bow64: ELF 64-bit LSB shared object

x86-64

version 1 (SYSV)

dynamically linked

interpreter /lib64/ld-linux-x86-64.so.2

for GNU/Linux 3.2.0

BuildID[sha1]=9503477016e8604e808215b4babb250ed25a7b99

not stripped

So if we now look at the assembler code, we see that the addresses are twice as big, and we have almost half of the instructions as with a 32-bit compiled binary.

Compile in 64-bit Format

student@nix-bow:~$ gdb -q bow64

Reading symbols from bow64...(no debugging symbols found)...done.

(gdb) disas main

Dump of assembler code for function main:

0x00000000000006bc <+0>: push rbp

0x00000000000006bd <+1>: mov rbp,rsp

0x00000000000006c0 <+4>: sub rsp,0x10

0x00000000000006c4 <+8>: mov DWORD PTR [rbp-0x4],edi

0x00000000000006c7 <+11>: mov QWORD PTR [rbp-0x10],rsi

0x00000000000006cb <+15>: mov rax,QWORD PTR [rbp-0x10]

0x00000000000006cf <+19>: add rax,0x8

0x00000000000006d3 <+23>: mov rax,QWORD PTR [rax]

0x00000000000006d6 <+26>: mov rdi,rax

0x00000000000006d9 <+29>: call 0x68a <bowfunc>

0x00000000000006de <+34>: lea rdi,[rip+0x9f]

0x00000000000006e5 <+41>: call 0x560 <puts@plt>

0x00000000000006ea <+46>: mov eax,0x1

0x00000000000006ef <+51>: leave

0x00000000000006f0 <+52>: ret

End of assembler dump.

However, we will first take a look at the 32-bit version of the vulnerable binary. The most important instruction for us right now is the call instruction. The call instruction is used to call a function and performs two operations:

1. it pushes the return address onto the stack so that the execution of the program can be continued after the function has successfully fulfilled its goal,
2. it changes the instruction pointer (EIP) to the call destination and starting execution there.

**GDB - Intel Syntax**

GDB - Intel Syntax

student@nix-bow:~$ gdb ./bow32 -q

Reading symbols from bow...(no debugging symbols found)...done.

(gdb) disassemble main

Dump of assembler code for function main:

0x00000582 <+0>: lea ecx,[esp+0x4]

0x00000586 <+4>: and esp,0xfffffff0

0x00000589 <+7>: push DWORD PTR [ecx-0x4]

0x0000058c <+10>: push ebp

0x0000058d <+11>: mov ebp,esp

0x0000058f <+13>: push ebx

0x00000590 <+14>: push ecx

0x00000591 <+15>: call 0x450 <\_\_x86.get\_pc\_thunk.bx>

0x00000596 <+20>: add ebx,0x1a3e

0x0000059c <+26>: mov eax,ecx

0x0000059e <+28>: mov eax,DWORD PTR [eax+0x4]

0x000005a1 <+31>: add eax,0x4

0x000005a4 <+34>: mov eax,DWORD PTR [eax]

0x000005a6 <+36>: sub esp,0xc

0x000005a9 <+39>: push eax

0x000005aa <+40>: call 0x54d <bowfunc> # <--- CALL function

<SNIP>

**Endianness**

During load and save operations in registers and memories, the bytes are read in a different order. This byte order is called endianness. Endianness is distinguished between the little-endian format and the big-endian format.

Big-endian and little-endian are about the order of valence. In big-endian, the digits with the highest valence are initially. In little-endian, the digits with the lowest valence are at the beginning. Mainframe processors use the big-endian format, some RISC architectures, minicomputers, and in TCP/IP networks, the byte order is also in big-endian format.

Now, let us look at an example with the following values:

* Address: 0xffff0000
* Word: \xAA\xBB\xCC\xDD

| **Memory Address** | **0xffff0000** | **0xffff0001** | **0xffff0002** | **0xffff0003** |
| --- | --- | --- | --- | --- |
| Big-Endian | AA | BB | CC | DD |
| Little-Endian | DD | CC | BB | AA |