Exploitation – Assembly and GDB Refresher

1 AT&T Assembly Syntax

Quelle: http://csiflabs.cs.ucdavis.edu/~ssdavis/50/att-syntax.htm (slightly modified)
Author: vivek

This article is a 'quick-n-dirty' introduction to the AT&T (and INTEL) assembly language syntax. Due to its simplicity, I use the NASM (Netwide Assembler) variant of the INTEL-syntax to cite differences between the formats.

1.1 The Basic Format

The structure of a program in AT&T-syntax is similar to any other assembler-syntax, consisting of a series of directives, labels, instructions - composed of a mnemonic[1] followed by a maximum of three operands. The most prominent difference in the AT&T-syntax stems from the ordering of the operands.

For example, the general format of a basic data movement instruction in INTEL-syntax is,

```
mnemonic       destination, source
```

whereas, in the case of AT&T, the general format is

```
mnemonic       source, destination
```

To some (including myself), this format is more intuitive. The following sections describe the types of operands to AT&T assembler instructions for the x86 architecture.

IMPORTANT: In GDB, you can switch between the two syntax versions by entering:

```
set disassembly-flavor intel  //the default
set disassembly-flavor att
```

1.2 Registers

All register names of the IA-32 architecture must be prefixed by a '%' sign, eg. %al,%bx, %ds, %cr0 etc.

```
mov    %ax, %bx
```

The above example is the mov instruction that moves the value from the 16-bit register AX to 16-bit register BX.

1.3 Literal Values

All literal values must be prefixed by a '$' sign. For example,

```
mov    $100,  %bx
mov    $A,    %al
```

The first instruction moves the value 100 into the register BX and the second one moves the numerical value of the ascii A into the AL register. To make things clearer, note that the below example is not a valid instruction,

```
mov   %bx,  $100
```

as it just tries to move the value in register bx to a literal value. It just doesn't make any sense.

[1] The keyword for this instruction. E.g., `mov` for a move instruction or add for doing an addition.

1.4 Memory Addressing

In the AT&T Syntax, memory is referenced in the following way,

```
segment-override:signed-offset(base,index,scale)
```

parts of which can be omitted depending on the address you want.

```
%es:100(%eax,%ebx,2)
```

Please note that the offsets and the scale should not be prefixed by '$'. A few more examples with their equivalent NASM-syntax, should make things clearer,

```
GAS syntax                              NASM memory operand
------------------                      -------------------
100                                     [100]
%es:100                                 [es:100]
(%eax)                                  [eax]
(%eax,%ebx)                             [eax+ebx]
(%ecx,%ebx,2)                           [ecx+ebx*2]
(,%ebx,2)                               [ebx*2]
-10(%eax)                               [eax-10]
%ds:-10(%ebp)                           [ds:ebp-10]
```

Example instructions,

```
mov     %ax,    100
mov     %eax,   -100(%eax)
```

The first instruction moves the value in register AX into offset 100 of the **data segment register (by default)**, and the second one moves the value in eax register to [eax-100].

1.5 Operand Sizes

At times, especially when moving literal values to memory, it becomes necessary to specify the size-of-transfer or the operand-size. For example the instruction,

```
mov    $10,   100
```

only specifies that the value 10 is to be moved to the memory offset 100, but not the transfer size. In NASM this is done by adding the casting keyword byte/word/dword etc. to any of the operands. In AT&T syntax, this is done by adding a suffix - b/w/l - to the instruction. For example,

```
movb   $10,   %es:(%eax)
```

moves a byte value 10 to the memory location [ea:eax], whereas,

```
movl   $10,   %es:(%eax)
```

moves a long value (dword) 10 to the same place. A few more examples,

```
movl    $100, %ebx
pushl   %eax
popw    %ax
```

1.6 Control Transfer Instructions

The `jmp`, `call`, `ret`, etc., instructions transfer the control from one part of a program to another. The `call` and `ret` instructions are typically used when transferring control from one function to another. `call` redirects control flow to a target location and `ret` jumps back to the instruction that immediately follows that `call`. However, the CPU needs to store the location where `ret` should jump back to somewhere. On x86 this is done on the stack (see also Appendix Inspecting the stack2.9). `call` will push the instruction pointer EIP to the stack before jumping to the target location. `ret` will pop EIP from the stack to jump back. This means that a program can jump back from a call to any location by modifying the saved return address on the stack. Generally speaking,

```
push %eax
ret
```

is a way of doing

```
jmp *%eax.
```

These instructions can be classified as control transfers to the same code segment (near) or to different code segments (far). The possible types of branch addressing are - relative offset (label), register, memory operand, and segment-offset pointers.

Relative offsets, are specified using labels, as shown below.

```
label1:
        …
  jmp   label1
```

Branch addressing using registers or memory operands must be prefixed by a '`*`'. To specify a "far" control tranfers, a '`l`' must be prefixed, as in '`ljmp`', '`lcall`', etc. For example,

```
GAS syntax                      NASM syntax
===========                     ===========
jmp     *100                    jmp  near [100]
call    *100                    call near [100]
jmp     *%eax                   jmp  near eax
jmp     *%ecx                   call near ecx
jmp     *(%eax)                 jmp  near [eax]
call    *(%ebx)                 call near [ebx]
ljmp    *100                    jmp  far  [100]
lcall   *100                    call far  [100]
ljmp    *(%eax)                 jmp  far  [eax]
lcall   *(%ebx)                 call far  [ebx]
ret                             retn
lret                            retf
lret $0x100                     retf 0x100
```

Segment-offset pointers are specified using the following format:

```
jmp    $segment, $offset
```

For example:

```
jmp    $0x10, $0x100000
```

2 Debugging programs with GDB/PEDA

Note that this guide applies to:

32-bit Linux binaries compiled using the gcc compiler and following the cdecl calling convention. See Section 2.6 for details on cdecl.

Binaries that have debug information attached and have not been compiled with options like `-fomit-frame-pointer`, `-finline-functions` or other options that might change the way functions are called and/or that affect the layout of the stack in some way.

On a 64-bit Linux, you must make sure that you can compile (`-m32` flag) and run 32-bit binaries. With 64-bit binaries, the disassembly of the sample code would look different because register use and naming as well as the set of available instructions differs. Furthermore, 64-bit architectures use different default calling conventions, for x86_64 it is the System V AMD64 ABI[2] calling convention.

A brief overview of the most relevant changes of the x86_x64 architecture over the x86 (cdecl) one can be found here: http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/

Use the *guide.c* program for this guide on how to debug programs with GDB/PEDA:

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

int last;

int add(int a, int b) {
   last = a + b;
   return a + b;
}

int main(int argc, char** argv) {
    printf("%d\n", add(1 ,2));
    printf("%d\n", last);
    return 1;
}
```

Compile guide.c with the following command:

```
gcc -g -m32 -no-pie -o guide guide.c
```

Note that when compiling without the –g option, analysing the binary with a debugger gets a lot harder since no debug information is added[3]. Furthermore, keep in mind that whenever GDB/PEDA outputs are shown, they might differ from what you see on your system. This concerns mainly addresses and ordering of content in memory (and the binary).

2.1 Starting GDB/PEDA

If GDB and the Python Exploit Development Assistance (PEDA) for GDB has been installed, you can start GDB/PEDA to debug the program named `guide` by entering:

```
gdb guide
```

2.2 Listing functions in a binary

To get information about the binary, you can list the functions in it using the command:

```
(gdb) info functions
```

[2] See http://www.x86-64.org/documentation/.

[3] See https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html for details

This results in the following output:

```
All defined functions:

File guide.c:
int add(int, int);
int main(int, char **);

Non-debugging symbols:
0x080482ac  _init
0x080482e0  printf@plt
0x080482f0  __libc_start_main@plt
0x08048310  _start
0x08048340  __x86.get_pc_thunk.bx
0x08048350  deregister_tm_clones
0x08048380  register_tm_clones
0x080483c0  __do_global_dtors_aux
0x080483e0  frame_dummy
0x08048495  __x86.get_pc_thunk.ax
0x080484a0  __libc_csu_init
0x08048500  __libc_csu_fini
0x08048504  _fini
```

2.3 Setting breakpoints

Breakpoints are used to temporarily stop the program at a designated location. Such a location can be a function name like main for the main function:

```
(gdb) break main

Breakpoint 1 at 0x8048440: file guide.c, line 13.
```

Or it can be a source line in the source code:

```
(gdb) break guide.c:8

Breakpoint 2 at 0x8048420: file guide.c, line 8.
```

Or it can be an address like e.g., the address of the function _start as shown in 2.2:

```
(gdb) break *0x08048310

Breakpoint 3 at 0x08048310
```

2.4 Running a program

To run the program simply use `run <args>`:

```
(gdb) run

Starting program: /home/hacker/exploitation/guide
[----------------------------------registers-----------------------------------]
EAX: 0xf7ffd918 --> 0x0
EBX: 0xf7ffd000 --> 0x23f3c
ECX: 0xffffdb54 --> 0xffffdc9d ("/home/hacker/exploitation/guide")
EDX: 0xf7fe88b0 (push   ebp)
ESI: 0xffffdb5c --> 0xffffdcbd ("LC_ALL=en_US.UTF-8")
EDI: 0x8048310 (<_start>: xor    ebp,ebp)
EBP: 0x0
ESP: 0xffffdb50 --> 0x1
EIP: 0x8048310 (<_start>: xor    ebp,ebp)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x804830a:             add    BYTE PTR [eax],al
   0x804830c:             add    BYTE PTR [eax],al
   0x804830e:             add    BYTE PTR [eax],al
=> 0x8048310 <_start>:    xor    ebp,ebp
   0x8048312 <_start+2>:  pop    esi
   0x8048313 <_start+3>:  mov    ecx,esp
   0x8048315 <_start+5>:  and    esp,0xfffffff0
   0x8048318 <_start+8>:  push   eax
```

```
[----------------------------------stack-------------------------------------]
0000| 0xffffdb50 --> 0x1
0004| 0xffffdb54 --> 0xffffdc9d ("/home/hacker/exploitation/guide")
0008| 0xffffdb58 --> 0x0
0012| 0xffffdb5c --> 0xffffdcbd ("LC_ALL=en_US.UTF-8")
0016| 0xffffdb60 --> 0xffffdcd0 ("LS_COL-
ORS=di=32:ln=35;40:so=32;40:pi=33;40:ex=93:bd=34;46:cd=34;43:su=0;41:sg=0;46:tw=0;4
2:ow=0;43:*.pdf=33:*.conf=94:*.sh=93:*.pl=93:")
0020| 0xffffdb64 --> 0xffffdd57 ("_=/usr/bin/gdb")
0024| 0xffffdb68 --> 0xffffdd66 ("LANG=en_US.UTF-8")
0028| 0xffffdb6c --> 0xffffdd77 ("LESS=-R")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 3, 0x08048310 in _start ()
```

Thanks to PEDA, GDB outputs a lot of information about the context of the program. There are three sections in the output after a breakpoint has been reached:

The first section of the output shows information about the content of the registers used. If a register contains a value that might be a valid memory address, PEDA tries to interpret/display the data at this memory location (smart dereferencing). For example, the ECX register seems to refer to a memory location containing a string and the EDI/EIP/EDX registers seem to refer to memory locations containing code.

The second section shows the disassembled code around the current instruction pointer (EIP).

The third section shows the content of the stack.

This information can also be displayed using the following commands:

```
context reg                //for the registers and flags
context code  [<NUMBER>]   //shows <NUMBER> of instructions around EIP
context stack [<NUMBER>]   //show the stack (<NUMBER>: # of WORDs to show)
context all                //shows reg, code and stack command outputs
```

2.5 Stepping

Once a program has stopped at a breakpoint you can continue to the next breakpoint with `continue`:

```
(gdb) continue

Continuing.
...
Breakpoint 1, main (argc=1, argv=0xbffff0d4) at guide.c:13
13  printf("%d\n", add(1 ,2));

(gdb) continue

Continuing.
...
Breakpoint 2, add (a=0x1, b=0x2) at guide.c:8
8      last = a + b;
```

You can also single-step instructions with `si`.

2.6 Showing locals and function arguments

Once we have reached a breakpoint, we can ask for information about all local variables using the command `info locals`.

To do so, let us start debugging the guide program from scratch by exiting GDB and restarting it:

```
(gdb) quit

gdb ./guide
```

and then setting the breakpoint as follows:

```
(gdb) break add
```

```
Breakpoint 1 at 0x8048418: file guide.c, line 8.
```

Now run the program and inspect the locals:

```
(gdb) run
```

```
Starting program: /home/hacker/exploitation/guide
Breakpoint 1, add (a=1, b=2) at guide.c:8
8   last = a + b;
```

```
(gdb) info locals
```

```
No locals.
```

The program does not have any local variables in the function `add` as GDB tells us. However, we can make it list the arguments:

```
(gdb) info args
```

```
a = 0x1
b = 0x2
```

2.7 Inspecting variables and registers

Let us assume you are now interested in the value of the variable `last` and where it is located:

```
(gdb) p last
```

```
$1 = 0
```

```
(gdb) p &last
```

```
$2 = (int *) 0x804a020 <last>
```

The value of `last` is 0 and its address is `0x804a020`. You can also inspect the values of registers using the command `p $<register name>`. To print the content of the ESP register, enter:

```
(gdb) p $esp
```

```
$3 = (void *) 0xbffffda90
```

However, it is probably more convenient to use the `context reg` command from PEDA.

2.8 Watchpoints - Breaking when a value of a variable changes

If you want to know, when and where our program changes the value of our variable `last`, you can achieve this with a watchpoint. When the value of a watched variable changes, the program will pause:

From 2.7 we know the address of the variable `last` and can therefore set the watchpoint for our sample program as follows:

```
(gdb) watch *0x804a020
```

```
Hardware watchpoint 2: *0x804a020
```

Restarting the program now stops when the value changes:

```
(gdb) run
```

```
Starting program: /home/hacker/exploitation/guide
...
Breakpoint 1, add (a=0x1, b=0x2) at guide.c:8
8   last = a + b;
```

```
(gdb) continue
```

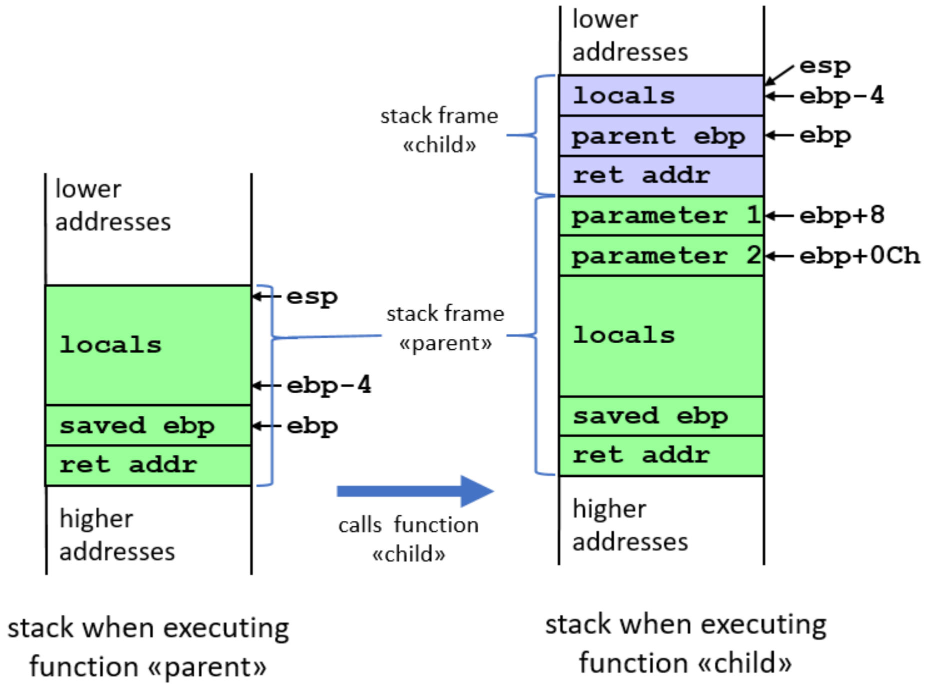
```
Continuing.
...
Hardware watchpoint 2: *0x804a020
Old value = 0x0
New value = 0x3
add (a=0x1, b=0x2) at guide.c:9
9   return a + b;
```

GDB will tell you the old and new value as well as where the change happened in the source code. That is not yet all you can do to inspect a program's state at a breakpoint.

Another interesting piece of information is the stack.

2.9 Inspecting the stack

To understand what's happening on the stack, remember that for every function call made, a stack frame is created to manage the resources of this function. In the following, we look at the stack of x86 (32-bit) binaries compiled with GCC with the **cdecl** calling convention (pronounced "see-deck-'ll", rhymes with "heckle")[4]. It is the default calling convention for x86 C compilers, although many compilers provide options to change the calling conventions used. For this system, the image below shows how the stack frame of an arbitrary function (parent) looks like. Furthermore, it shows how the stack changes when the parent function calls and executes another function (child). The called function has two parameters:



To locate the top and bottom of the current stack frame, the two pointers (CPU registers) ESP and EBP are the key. The stack pointer (ESP) point always to the topmost entry on the stack. It is used to indicate where data shall be pushed onto the stack or popped from the top of the stack. The base pointer (EBP) always points to the bottom of the currently used stack frame (not exactly the bottom, but one position above the return address). All x86 calling conventions define EBP as being preserved[5] across function calls. It remains constant even if the stack frame grows and shrinks and is therefore

[4] Different compilers and/or architectures (e.g., x86 and x86_64) might have a different understanding (layout, start/end) of a stack frame and compiler options like `-fomit-frame-pointer` for GCC might impact it too.

[5] However, this might not be true when a compiler generates (heavily) optimized code using flags like `-fomit-frame-pointer` or function inlining heuristics included in optimization levels like O2 or O3.

used to relatively address all other positions in the current stack frame. This enables stack walking in a debugger and viewing of other frame's local variables.

Furthermore, there is the instruction pointer (EIP) which always points to the instruction to be executed next. When a function calls another function, the EIP is modified to point to the code of the called function. To remember where to continue the execution of the current function, the value of EIP is pushed to the stack (ret addr) before it is modified.

Let's now look a bit in more details into what happens when a function (parent) calls another function (child) and how this affects the stack:

1. If the function to be called has parameters, they are placed on the stack. The parameters belong to the stack frame of the caller (parent).
2. A `call` instruction is executed which leads to the creation of a new stack frame:
3. It pushes the content of the EIP register onto the stack (ret addr). Note that the push instruction adjusts the stack-pointer accordingly.
4. The address provided to the call instruction is loaded into EIP and execution continues from there.
5. The function **prologue** of the called function is executed. It consists of a few lines of code at the start of the function that prepare the stack and registers for use within the function.
6. It pushes the EBP onto the stack to save the EBP of the calling function.
7. The current value of ESP is stored in EBP – it marks the base of the new stack frame
8. Space for local variables is allocated (if any) by altering the ESP accordingly.
9. The function performs its work.

The **epilogue** of the function is executed – it restores the stack and registers to their state before the function was called. The epilogue often contains the `leave` instruction followed by a `ret` instruction. If the `leave` instruction is not used, there should be multiple other instructions with the same result.

`leave` sets the ESP to EBP so that it points to the *saved EBP* and then it restores the EBP from the parent function by popping the *saved EBP* from the stack. Hence, we end up with ESP pointing to the return address.

`ret` pops the return address off the stack and into EIP so that the program continues execution from just after where the original call was made. After popping the return address, ESP points to the top of the stack of the stack frame of the parent function.

Note that Integer values and memory addresses are returned in the EAX and floating point values in the ST0 x87 register[6], that EAX, ECX, and EDX are caller-saved, and the rest callee-saved. Callers must save EAX, ECX or EDX only if the content of the register is expected to be the same after the call. Callees must save only registers whose content they overwrite.

It is now time to look at the stack frame information of our *guide* program. Exit GDB and then enter:

```
gdb guide

(gdb) break add
Breakpoint 1 at 0x8048418: file guide.c, line 8.

(gdb) run
…

(gdb) context stack 20
…
```

[6] The x87 floating point registers ST0 to ST7 must be empty (popped or freed) when calling a new function, and ST1 to ST7 must be empty on exiting a function. ST0 must also be empty when not used for returning a value.

You should see the following information about the stack:

Stack frame of function *add*

```
0000| 0xffffda90 --> 0xffffdaa8              //saved EBP from main
0004| 0xffffda94 --> 0x8048455               //saved return address
```

Stack frame of function *main*:

```
0008| 0xffffda98 --> 0x1                     //first parameter for function add
0012| 0xffffda9c --> 0x2                     //second parameter for function add
0016| 0xffffdaa0 --> 0xffffdac0 --> 0x1
0020| 0xffffdaa4 --> 0x0
0024| 0xffffdaa8 --> 0x0
0028| 0xffffdaac --> 0xf7e0e276 (<__libc_start_main+246>:  add    esp,0x10)
0032| 0xffffdab0 --> 0x1
0036| 0xffffdab4 --> 0xf7fa9000 --> 0x1b2db0
0040| 0xffffdab8 --> 0x0                     //saved EBP
0044| 0xffffdabc --> 0xf7e0e276              //saved return address
```

Pre-main stack:

```
0048| 0xffffdac0 --> 0x1
0052| 0xffffdac4 --> 0xffffdb54 --> 0xffffdc9c ("/home/hacker/exploitation/guide")
…
```

From the output we can see, that it matches the basic stack layout described before. The first address listed `0xffffda90` (=offset 0000) is the lowest memory address currently in use by the stack. This corresponds to the top of the stack, since the stack grows from high to low memory addresses. Hence, this should correspond to the content of ESP (stack pointer). Checking this with

```
(gdb) context reg
```

```
EBP: 0xffffda90
…
ESP: 0xffffda90
```

shows that this is true. Since nothing has been pushed on the stack by the add method yet, ESP has the same value as EBP. You can also ask GDB to output some summary information about the current stack frame:

```
(gdb) info frame
```

```
Stack level 0, frame at 0xffffda98:
 eip = 0x8048418 in add (guide.c:8); saved eip = 0x8048455
 called by frame at 0xffffdac0
 source language c.
 Arglist at 0xffffda90, args: a=0x1, b=0x2
 Locals at 0xffffda90, Previous frame's sp is 0xffffda98
 Saved registers:
  ebp at 0xffffda90, eip at 0xffffda94
```

This tells you what the start of the stack frame is: **`0xffffda98`**. It seems that the start is always set to ebp+8 for x86 cdecl therewith pointing to the first byte NOT belonging to the stack frame anymore. Byte **`0xffffda98`** is the first byte of parameter `a`.

Next, you can see the current value of EIP **`0x8048418`**, the Extended Instruction Pointer, which points to the instruction executed next. The corresponding line in the source code (*guide.c:8*) is listed when compiling with “-g” for inclusion of debug information. The current value of the EIP register is followed by the value of the **saved EIP** (**`0x8048455`**). The saved EIP is where the program will jump to on return of the current function. Other information that can be found is the location of arguments and local variables. Unfortunately, this information is **not correct**; they cannot be located at the same location (**`0xffffda90`**). This seems to be a known bug in GDB[7].

[7] https://sourceware.org/bugzilla/show_bug.cgi?id=13260

However, it is easy to find locals and arguments based on the current value of EBP and knowledge of the stack layout:

Locals can be found after EBP (**`0xffffda90`**) toward lower addresses (here: none)

Arguments can be found (if any) at EBP+8 (**`0xffffda98`**) toward higher addresses.

If you don't trust the visualization from PEDA, you can verify the memory itself starting from ESP upwards. You can print four words (=32-bit numbers) in hexadecimal format by entering:

```
(gdb) x/4wx 0xffffda90
```

`0xbffffda90:`	`0xffffdaa8`	`0x08048455`	`0x00000001`	`0x00000002`
	saved EBP. EBP of previous frame	saved EIP := return adress.	argument `a` at offset 0x8 from EBP	argument b at offset 0xC from EBP

Note that when printing in the word format, GDB translates the effective byte values stored in little-endian format to the stored 32-bit number value. Hence, in the above printout, the fourth byte with value (`0xa8`) is the byte stored at address `0xffffda90`. This can be checked when printing the same memory bytes in the byte format (16bx = print 16 bytes as hex numbers):

```
(gdb) x/16bx 0xffffda90

0xffffda90:0xa8    0xda    0xff    0xff    0x55    0x84    0x04    0x08
0xffffda98:0x01    0x00    0x00    0x00    0x02    0x00    0x00    0x00
```

2.10 Disassembling the program

To look at the assembly output of a function use the `pdisas` command:

```
(gdb) pdisas add

Dump of assembler code for function add:
   0x0804840b <+0>:       push   ebp
   0x0804840c <+1>:       mov    ebp,esp
   0x0804840e <+3>:       call   0x8048495 <__x86.get_pc_thunk.ax>
   0x08048413 <+8>:       add    eax,0x1bed
   0x08048418 <+13>:      mov    ecx,DWORD PTR [ebp+0x8]
   0x0804841b <+16>:      mov    edx,DWORD PTR [ebp+0xc]
   0x0804841e <+19>:      add    edx,ecx
   0x08048420 <+21>    :  mov    eax,0x804a020
   0x08048426 <+27>:      mov    DWORD PTR [eax],edx
   0x08048428 <+29>:      mov    edx,DWORD PTR [ebp+0x8]
   0x0804842b <+32>:      mov    eax,DWORD PTR [ebp+0xc]
   0x0804842e <+35>:      add    eax,edx
   0x08048430 <+37>:      pop    ebp
   0x08048431 <+38>:      ret
```

On the first line, EBP is pushed to the stack. Next, the call to *_x86.get_pc_thunk.ax* is related to logic used when code is compiled in a position independent way. It loads the position of the code into the EAX register, which allows global objects (which have a fixed offset from the code) to be accessed as an offset from that register. However, this is not used in the code above. Next, the arguments are loaded into the ECX and EDX registers. Then, an addition is done and the result is written to `0x804a020`, which corresponds to the address of variable `last`:

```
(gdb) p &last

$3 = (int *) 0x804a020 <last>
```

Finally, the same addition is carried out another time, EBP is popped and the function returns.

2.11 Altering memory

With GDB we can also alter values in memory.

```
gdb guide
…

(gdb) break guide.c:9
Breakpoint 1 at 0x8048428: file guide.c, line 9.

(gdb) run
Starting program: /home/hacker/exploitation/guide
…
Breakpoint 1, add (a=0x1, b=0x2) at guide.c:9
9       return a + b;

(gdb) p a
$1 = 1

(gdb) p &a
$2 = (int *) 0xffffda98

(gdb) set {int} 0xffffda98 = 9911

(gdb) p a
$3 = 0x26b7
```

2.12 Quiz

Compile *quiz-a.c* as follows:

```
gcc -m32 -g -o quiz-a quiz-a.c
```

```c
int foo(int a, int b) {
   int c = 3;
   return a + b + c;
}

int main(int argc, char** argv) {
   return foo(2, 3);
}
```

Code Sample 1 – quiz-a.c

Open it in GDB and set a breakpoint at the start of the function `foo`. Next, do the following:

Find the address to which `foo` will return to (=saved EIP).

Find the address of the saved EIP on the stack.

Modify the saved EIP to point to NULL (with GDB) then let the program continue.

Question: What is the value of the saved EIP and what happens after you let the program continue? And why does it happen?

```
(gdb) break foo
(gdb) run
(gdb) info frame
=> eip at = 0xffffdaac

(gdb) set {int} 0xffffdaac = 0x0
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
```

Answer:

-Value of saved EIP: 0x5655559c.
-Segmentation fault: It happens because when the Instruction Pointer points to null, the attempt to read this "memory address" leads to a segmentation fault.

Next, inspect the assembly code of the `foo` function and find out how much stack space GCC allocates. Furthermore, set a breakpoint just before the first addition in `foo` and calculate the difference between the address pointed to by ESP and EBP.

Question: How much stack space does the compiler allocate and at which line in the assembly code?

```
(gdb) disas foo
Dump of assembler code for function foo:
   0x56555560 <+0>:    push   ebp
   0x56555561 <+1>:    mov    ebp,esp
   0x56555563 <+3>:    sub    esp,0x10                                   //the compiler allocates 16 bytes (0x10 hex)
   0x56555566 <+6>:    call   0x565555a1 <__x86.get_pc_thunk.ax>
   0x5655556b <+11>:   add    eax,0x1a95
   0x56555570 <+16>:   mov    DWORD PTR [ebp-0x4],0x3
   0x56555577 <+23>:   mov    edx,DWORD PTR [ebp+0x8]
   0x5655557a <+26>:   mov    eax,DWORD PTR [ebp+0xc]
   0x5655557d <+29>:   add    edx,eax
   0x5655557f <+31>:   mov    eax,DWORD PTR [ebp-0x4]
   0x56555582 <+34>:   add    eax,edx
   0x56555584 <+36>:   leave
   0x56555585 <+37>:   ret
End of assembler dump.
```

Question: What are the values in ESP and EBP and does EBP – ESP make sense?

EBP: 0xffffdaa8
ESP: 0xffffda98

=> EBP – ESP = 16 Bytes => Yes, makes sense. Is the amount of bytes the stack is enlarged by the code line identified above.