TD 4: x86-32 and x86-64 assembly (Part 2)

1 Compiler's generated assembler

1. Compile the following C code and analyze the generated assembler code (use the '`gcc -no-pie -S -m32`' option). Try to locate the position of the variables `i`, `j` and the call to the function `lrint()` and the way its argument is passed.

```c
#include <math.h>
int main () {
        long i = 10, j, k = 7;
        i *= 255;
        j = lrint (log (i) * i);
        return j;
}
```

2. Compile it with the following commands to have the i386 (`-m32`) and amd64 (`-m64`) vesions and the different optimization levels (`-O0` no optimization, `-O3` all optimizations, `-Os` optimize for space), and look at the the result of '`objdump -Sd`' to see the differences in the `main` function :
   - `gcc -no-pie -Wall -Wextra -g -std=c99 -m32 -O0 -lm`
   - `gcc -no-pie -Wall -Wextra -g -std=c99 -m32 -O3 -lm`
   - `gcc -no-pie -Wall -Wextra -g -std=c99 -m32 -Os -lm`
   - `gcc -no-pie -Wall -Wextra -g -std=c99 -m64 -O0 -lm`
   - `gcc -no-pie -Wall -Wextra -g -std=c99 -m64 -O3 -lm`
3. Take the following C program that calls the function `foo()` from `main()`. Then, use `objdump` to look at the generated assembler in i386 and amd64.

```c
int foo (int x) {
        int i = 10;
        return i+2;
}

int main () {
        return foo (1);
}
```

4. Add several arguments to the function `foo()` (as below) and look at the generated code in i386 and amd64 assembler. For the amd64 code, find the maximum number of arguments (integers and floats) after which it starts using the stack.

```c
void foo (int a, int b, int c, int d, int e, int f, int g) {
        int i = 10 + a + b + c + d + e + f + g;
}

int main () {
        foo (1, 2, 3, 4, 5, 6, 7);
```

```
        return 0;
}
```

2 Installing pframe in your environment

Get pframe with
`wget https://dept-info.labri.fr/~thibault/SecuLog/pframe.tgz`
Unpack it into your home :

```
tar -C ~ -xf pframe.tgz
```

Add this at the end of your `.bashrc` file in your home :

```
export PYTHONPATH=$HOME/pframe${PYTHONPATH:+:${PYTHONPATH}}
```

Add this to your `.gdbinit` file (create it if it does not exist in your home yet) :

```
python import pframe
```

Close your terminal, re-open it (so `.bashrc` is reloaded) Your `gdb` now has a `pframe` command!
Note : if you are installing on your own machine, install the `python3-dbg` package.

3 Looking at the stack

1. Compile the following program **with**
   `-m32 -no-pie -O0 -g`
   and check out how values are passed : for each `foo` function, in `gdb`, set a breakpoint on a callee function, let `gdb` stop on it, type `pframe`, and look at the caller and callee code, to check how all this is layed out on the stack ; draw a picture of it on paper.
2. Also check out how this shows up in 64bit with `-m64`.

```
int foo1(int t[2]) {
        return t[0]+t[1];
}

struct s {
        int f0;
        int f1;
};
int foo2(struct s *f) {
        return f->f0 + f->f1;
}

int foo3(struct s f) {
        return f.f0 + f.f1;
}

struct s2 {
        int f[2];
};

int foo4(struct s2 f) {
        return f.f[0] + f.f[0];
```

```
}

int t[2] = {1,2};

int main () {
        foo1(t);
        struct s mys = { 1, 2 };
        foo2(&mys);
        foo3(mys);

        struct s2 mys2 = { { 1, 2 } };
        foo4(mys2);

        return 0;
}
```

3. Use the structure struct `mystruct { int a; int* b; double c; };` in the program as below. And, look at the generated assembler (both i386 and amd64) of the function `foo()`.

```
struct mystruct {
        int a;
        int* b;
        double c;
};

struct mystruct foo (int a, int *b, double c) {
        return (struct mystruct) { a, b, c };
}

int main () {
        int i;
        struct mystruct result = foo (1, &i, 3.0);
        return 0;
}
```

4. Write an assembler program (in i386 and amd64) which calls the `puts("Hello World!")` function from the libc.
5. Write an assembler program (in i386 and amd64) which calls the `gets` function from the libc, then `puts` to print back the same text.

4 Advanced problem

What does this function compute?

```
000006a0 <mystery>:
6a0:   55              push %ebp
6a1:   31 d2           xor  %edx,%edx
6a3:   31 c0           xor  %eax,%eax
6a5:   b9 01 00 00 00  mov  $0x1,%ecx
6aa:   89 e5           mov  %esp,%ebp
6ac:   53              push %ebx
6ad:   31 db           xor  %ebx,%ebx
6af:   3b 55 08        cmp  0x8(%ebp),%edx
6b2:   7d 11           jge  6c5 <mystery+0x25>
```

```
6b4:   83 fa 01        cmp  $0x1,%edx
6b7:   89 d0           mov  %edx,%eax
6b9:   7e 07           jle  6c2 <mystery+0x22>
6bb:   8d 04 0b        lea  (%ebx,%ecx,1),%eax
6be:   89 cb           mov  %ecx,%ebx
6c0:   89 c1           mov  %eax,%ecx
6c2:   42              inc  %edx
6c3:   eb ea           jmp  6af <mystery+0xf>
6c5:   5b              pop  %ebx
6c6:   5d              pop  %ebp
6c7:   c3              ret
```