Team Details:-

Rohan Gyani: 110040001
Sahil Jindal: 110020043
Abhishek Gupta: 110040067
Mridul Jain: 110040083

All the experiments were conducted on intel x86 machine or x86-64 machine with Linux(Ubuntu) as OS.

Progress Done:-

1. Studied Intel Instruction Set:-
    a. Registers are preceded by %, and constants are preceded by $.
    b. Register operands in 64-bit instructions begin with r, like %rsp, whereas, in 32-bit instructions register operands begin with e, like %eax.
    c. 32-bit instructions end with 'l', whereas 64 bit instructions end with 'q'.
    d. The destination register is the second operand as opposed to first operand in mips.
    e. ALU instructions have max. of 2 operands instead of 3 in MIPS.
    eg: subq $48, %rsp
    Description:- %rsp=%rsp-48
    f. Addressing is done in similar way as in MIPS.
    eg: -8(%rsp).
    g. We studied the assembly code generated by compiling a c-program on x86 and x86-64 both.
    h. Some useful instructions:-
        i. call: pushes address of next instruction on stack and transfer controls to address specified by the operand of the current instruction.
        ii. leave: adjust the stack pointer and the frame pointer such that they will contain correct values w.r.t caller function.
        iii. ret: pops return address from the stack and jumps to it.

2. Stack Structure Examination:-
    a. As is in MIPS, stack grows from higher memory address to lower address.
    b. In x86-64, %rsp is used to store the stack pointer and %rbp is used to store the frame pointer.
    c. When a subroutine is called the stack looks like the following:-

    For x86:- when a subroutine is called, the parameters are passed on to the stack straight away, i.e. the caller function stores them directly into the stack.
        i. At the bottom, are the parameters which are passed to this function
        ii. On top of them is the return address.
        iii. Above this, frame pointer for the previous function is stored and will be retrieved and used when the function will return to the caller function.

iv. On top of them, we've local variables defined in this function.

For x86-64:- Here upto 6 parameters are passed via registers and the remaining are store on the stack by the caller function

1) At the bottom are the parameters which were directly stored on the stack by the caller function.

2) On top of them is the return address.

3) Above this, frame pointer for the previous function is stored and will be retrieved and used when the function will return to the caller function.

4) On top of this, local variables defined in this function are stored

5) On top of this, the parameters which were passed to this function via registers are stored.

d. We observed in x86 that if even one integer variable was declared in a function, minimum of 16 bytes were allocated on the stack, but if no local variable was declared no extra space was allocated in the stack.

3. Canary Defence:-
    a. Some bytes are read from the memory and are stored in the neighbourhood of the return address of the function. When deallocating the stack frame of that particular function, we check that value using xor. If xor is not 0, it indicates that buffer overflow has occurred and __stack_chk_fail function will be called. If xor is 0, then ZF flag is set and using the je instruction (jump to the given label if ZF flag is 1) we skip the __stack_chk_fail function and the normal execution continues (ZF flag, i.e. zero flag, is set to 1 if the result of arithmetic operation is zero).
    b. Flag to disable canary defence mechanism:
    -fno-stack-protector
    c. As per our observation, defence mechanism is used when char array is used in our code but it was not needed when an int or an int array is used in the code. This may be because buffer overflow vulnerability could be exploited only in a char array.
    d. One strange observation, the order in which local variables are pushed on the stack is reversed as compared to when we're not using the above flag in (b) part. We're not yet able to determine the reason for this.

4. CFI Directives:-
    a. Can be disabled using the flag:
    -fno-asynchronous-unwind-tables
    b. Used to direct exception handling.
    c. The only purpose of using this flag was to make our assembly code neat and easier to read (as removing these directives did not affect our results).

What we plan to do further:-

1. Will try to change the return address of a function.
2. Will try to inject shell code into the array and point the return address to the beginning of the array so that the a new shell will be created and will have same privileges as the current program.
3. Will continue our study of the intel instruction set.

APPENDIX

Relevant Code(This is the assembly code of one of our C programs. We studied the intel instruction set as well as canary defense mechanism from such assembly codes)

```
.file   "temp3.c"
   .text
   .globl   func
   .type    func, @function
func:
   pushq    %rbp            // storing the frame pointer of the previous function
   movq     %rsp, %rbp   // making the frame pointer of this function equal to current stack pointer
   subq     $48, %rsp      // moving the stack pointer up by 48
   movl     %edi, -36(%rbp)      // storing the 3rd argument in the stack using offset and frame
pointer
   movl     %esi, -40(%rbp)     // storing the 2nd argument in the stack using offset and frame
pointer
   movl     %edx, -44(%rbp)    // storing the 1st argument in the stack using offset and frame
pointer

   movq     %fs:40, %rax
   movq     %rax, -8(%rbp)              // storing the word read from memory in stack
   xorl    %eax, %eax            // this is for making eax 0
   movq     -8(%rbp), %rax
   xorq     %fs:40, %rax                // comparing the word in the stack with the original word
present in the memory

 // The above instructions are for canary checking, here initially we stored a word from memory
into our stack and at the end of the program, we compare that stored word in stack with the
original word from the memory. If the xor is 0, then that word is intact, which means that our
return address will also be intact.
```

```
 je    .L2                                    // jump to L2 if zero flag is set
    call    __stack_chk_fail
.L2:
    leave
    ret
    .size    func, .-func
    .globl    main
    .type    main, @function
main:
    pushq    %rbp
    movq    %rsp, %rbp
    movl    $3, %edx              // storing the function parameters in the registers
    movl    $2, %esi
    movl    $1, %edi
    call    func
    movl    $0, %eax
    popq    %rbp
    ret
    .size    main, .-main
    .ident    "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
    .section    .note.GNU-stack,"",@progbits
```