Benton Guess
CSCE-451
Jan 27, 2019

Assignment Reflection

This assignment was mostly a review of what I had worked on over the break, however I still rewrote the code since there were some key changes in what I wrote over break and how I wrote code for this assignment that led to a stronger learning experience. First, the code I wrote over break was in Visual Studio 2017 MASM in x64-86, which despite having no major differences in terms of structure, was completely incompatible with NASM i386, which is the format I wanted to turn this assignment in. This ended up being very helpful since I had the chance to go over my old code and rewrite it after having gained some more tools for understanding assembly from class. What I found was that upon the second time writing the code I was producing more readable and logical code, and repeating far fewer lines and instructions. The most helpful part of this assignment was gaining a better understanding of the stack and how to use it in conjunction with subroutines. This was something I very much struggled with over the break, but now I feel much more comfortable with these concepts.

This assignment also helped me get more comfortable with the reverse engineering process, specifically the bonus where we could print something to the screen. To do this, I used Godbolt to compile basic C main functions into assembly that used printf and tried to figure out how to apply this to code I wrote in NASM but linked in GDB. This gave me a much stronger understanding of the calling process for C and how to interpret that from assembly code. It was also helpful throughout this entire assignment since most of the time I was trying to imagine how my assembly would translate to source code. Having this visualization of how to think about not just the original lines, but the overall logic of the source code has been helpful in lab and I imagine it will be helpful in the future assignments and lab.

Another huge benefit of this assignment was getting to understand the tools associated with reverse engineering better. I had never used Godbolt, but I found myself using it in this assignment to get an idea of how different high level logical processes look in assembly. I also enjoyed getting to know ”ld” and ”gcc” better as tools for linking and making use of arbitrary object files.

My main critique with this assignment was that there was too much of the same task occurring over and over again. Most of the problems were geared towards pushing an array onto the stack and then removing said array with some change to the process. If there was some variety in the types of tasks we had to accomplish, it would make the assignment more interesting and likely encourage us to explore more varied methods.

For improvements to this assignment, I would suggest that instead of writing this entirely in assembly, there would be some tasks entirely in NASM with some others writing extern functions for a language like C. I think by doing this we would not only learn about the process of writing assembly, but also see how functions are called and used in C relative to the underlying assembly. For example, in the style of the current assignment, we could have an array pointer passed to an assembly function from C, and we would do something like take the average or variance. This way, it would be easier to set up the program to do something more complex in C, we learn about assembly through writing a function that does something with input passed through C, and all of this would give us better insight into understanding how calls to functions are made. It would also be very simple to do this in C through linking the assembly with an extern function. This would be a great way for us to not only become more friendly with assembly, but also learn more about how C interfaces with its underlying functions.

Problem 1

```
global _start

section .data
section .text

_start:
   _problem1:
   mov ebp, esp
   mov eax, 0

   ; push to stack
   push 4
   push 77
   push 18
   push 57
   push 9

   ; sum the list of numbers
   pop edx
   add eax, edx
   pop edx
   add eax, edx
   pop edx
   add eax, edx
   pop edx
   add eax, edx
   pop edx
   add eax, edx


   ; divide the sum by the count of the list
   mov edx, 0
   mov ebx, 5
   idiv ebx
   ; Question 2 starts after this instruction
```

The code was executed as is assembled by ”nasm -f elf32 -g -F dwarf question1.s -o ex.o” and linked with ”ld -m elf_i386 ex.o -o ex”. Note that there is not an interrupt to exit the executable, since the value stored in eax will be used in question 2, which begins execution immediately after this function finishes running. The ”ld” command is run with the usual arguments, however the ”nasm” command uses the -g and -F flags with the argument dwarf. This made i

Problem 2

```
_problem2:
mov dword [ebp - 4], eax ; Store our initial avg on the stack (frees up
    the regs)
mov dword [ebp - 8], 20 ; Lower Bound for comp
mov dword [ebp - 12], 30 ; Upper Bound for comp
mov dword [ebp - 16], 0x7FFFFFFF ; Lower Range
mov dword [ebp - 20], 0x0 ; Upper Range
mov dword [ebp - 24], 0 ; Total for full range
mov dword [ebp - 28], 0 ; Number of increments for full sum
mov dword [ebp - 32], 0 ; Total for partial range
mov dword [ebp - 36], 0 ; NUmber of increments for partial sum
sub esp, 36

; Get everything on the stack
mov ecx, 1
LOAD_LOOP:
   push ecx
   inc ecx
   cmp ecx, [ebp - 4]
   jle LOAD_LOOP

; Do our operations in one loop
mov ecx, 1
ITER_LOOP:
   ; Pop the val into ebx
   pop ebx

   ; Normal sum stuff
   add [ebp - 24], ebx
   inc dword [ebp - 28]

   ; Calculates the partial sum
   cmp ebx, [ebp - 12]
   jge ADD_FOR_PARTIAL
   cmp ebx, [ebp - 8]
   jle ADD_FOR_PARTIAL
   jmp SKIP_FOR_PARTIAL
   ADD_FOR_PARTIAL:
      add [ebp - 32], ebx
      inc dword [ebp - 36]
   SKIP_FOR_PARTIAL:

   ; Calculates the range
   cmp ebx, [ebp - 16]
   jge NO_REPLACE_LOW
   mov [ebp - 16], ebx
   NO_REPLACE_LOW:
   cmp ebx, [ebp - 20]
   jle NO_REPLACE_HIGH
   mov [ebp - 20], ebx
   NO_REPLACE_HIGH:
        ; Compare for loop
        inc ecx
        cmp ecx, [ebp - 4]
        jle ITER_LOOP

    mov edx, 0

    ; Get Partial sum E[x]
    mov eax, [ebp - 32]
    idiv dword [ebp - 36]
    mov ebx, eax

    ; Get Full sum E[x]
    mov eax, [ebp - 24]
    idiv dword [ebp - 28]

    ; Put ranges in ecx and edx
    mov ecx, [ebp - 16]
    mov edx, [ebp - 20]

_end:
    mov ebx, 0
    mov eax, 1
    int 0x80
```

The code was executed as is assembled by ”nasm -f elf32 -g -F dwarf question1.s -o ex.o” and linked with ”ld -m elf_i386 ex.o -o ex”. Note that there is not a proper header since this code makes use of the output of problem 1 in eax, so therefore this code begins execution immediately after in the same binary.

Problem 3

```
; nasm -f elf32 -g -F dwarf question3.s -o ex.o
; ld -m elf_i386 ex.o -o ex

global _start

section .data
section .text

_start:
_problem3:
   mov ebp, esp
   ; Make a place to store the iterations, mult, and sum
   mov dword [esp - 4], 0 ; Store the iterations
   mov dword [esp - 8], 1 ; Mult lower 32
   mov dword [esp - 12], 0 ; mult upper 32
   mov dword [esp - 16], 0 ; Sum location
   mov dword [esp - 20], 100 ; initial loop limit
   sub esp, 24

   ; Load the numbers into the loop for 0:[epb - 20], count the iterations
      with esi
   mov ecx, 0
   LOOP_LOAD_100:
      push ecx
      ; Checks if our current number is equal to a constant we want to repeat
         10 times
      cmp ecx, 42
      jne INC_NUMS
         ; unrolled this part since there seems this repeats the exact same
            action
         mov dword [esp - 4], 42
         mov dword [esp - 8], 42
         mov dword [esp - 12], 42
         mov dword [esp - 16], 42
         mov dword [esp - 20], 42
         mov dword [esp - 24], 42
         mov dword [esp - 28], 42
         mov dword [esp - 32], 42
         mov dword [esp - 36], 42
         sub esp, 36
         add dword [ebp - 4], 9
      INC_NUMS:
      ; increment the loop variable and the iteration counter
      inc ecx
      inc dword [ebp - 4]
      cmp ecx, [ebp - 20]
      jle LOOP_LOAD_100

   ; prepare for another loop 1:[ebp - 20]
   mov ecx, 1
   mov esi, [ebp - 4]
```

```
    mov [ebp - 20], esi

    ; Mult numbers less than 30 and sum everything
    LOOP_THRU_100:
       pop ebx
       ; sum
       add [ebp - 16], ebx
       cmp ebx, 30
       jge NO_MULTIPLY_EBX
          ; multiply
          imul ebx, [ebp - 8]
          mov [ebp - 8], ebx
       NO_MULTIPLY_EBX:
       inc ecx
       cmp ecx, [ebp - 20]
       jle LOOP_THRU_100
    mov eax, [ebp - 8] ; Multiplication Result in eax
    mov ebx, [ebp - 16] ; Sum result in ebx
    mov ecx, [ebp - 4] ; Total number of iterations in ecx

_end:
    mov ebx, 0
    mov eax, 1
    int 0x80
```

The code was executed as is assembled by ”nasm -f elf32 -g -F dwarf question3.s -o ex.o” and linked with ”ld -m elf_i386 ex.o -o ex”.

Extra Credit: Encrypter with CLI

```
; nasm -f elf32 -g -F dwarf encrypter.s -o encrypter.o
; gcc -m32 -o encrypter encrypter.o -lc

global main

section .data
   enc:     db "Encrypted Value %d", 0x0a, 0x00
   dec:     db "Decrypted Value %d", 0x0a, 0x00
   failure: db "Arguments: [0 = encrypt| 1 = decrypt] [key] [value]", 0x0a,
      0x00

section .text
   extern printf
   extern atoi

; Randomizes a single byte value, passed over al
rng_gen:
   mov ah, al

   ; Seed the value
   add al, 0xb

   ; shift and xor to generate a random number
   shr al, 3
   xor ah, al
   mov al, ah
   shl al, 5
   xor ah, al
   mov al, ah
   shr al, 2
   xor ah, al
   mov al, ah

   ret

; Takes its value from what was already on the stack, does rand_gen for all 4
    bytes of
;     the passed 32 bit integer
rng_gen_4_bytes:
   push ebp
   mov ebp, esp
   push 0

   mov al, [ebp + 11]
   call rng_gen
   mov [esp+3], al

   mov al, [ebp + 10]
   call rng_gen
   mov [esp+2], al
```

```
    mov al, [ebp + 9]
    call rng_gen
    mov [esp+1], al

    mov al, [ebp + 8]
    call rng_gen
    mov [esp], al

    pop eax
    pop ebp
    ret

; Takes a key (which will be used as a decrypter as well)
encrypt:
    push ebp
    mov ebp, esp

    ; Gets the 4 random bytes using the seed on the stack
    push dword [ebp + 12]
    call rng_gen_4_bytes

    ; Adds and returns over eax
    add esp, 4
    mov ebx, eax
    mov eax, [ebp + 8]
    xor eax, ebx

    pop ebp
    ret

decrypt:
    push ebp
    mov ebp, esp

    ; Gets the 4 random bytes using the seed on the stack
    push dword [ebp + 12]
    call rng_gen_4_bytes

    ; Subtracts and returns over eax
    add esp, 4
    mov ebx, eax
    mov eax, [ebp + 8]
    xor eax, ebx

    pop ebp
    ret


main:
    mov ebp, esp
    sub esp, 12

    ; Ensures that we have the right number of arguments
    cmp dword [ebp + 4], 4
```

```
jne Failed_Code

; Parse command line args using atoi
; get type of op
mov edx, [ebp + 8]
mov eax, [edx + 4]
push eax
call atoi
add esp, 4
mov [ebp - 4], eax

; get key
mov edx, [ebp + 8]
mov eax, [edx + 8]
push eax
call atoi
add esp, 4
mov [ebp - 8], eax

; get value
mov edx, [ebp + 8]
mov eax, [edx + 12]
push eax
call atoi
add esp, 4
mov [ebp - 12], eax

; Check CLA 1 to see what operation is preferred, error out if its nonsense
mov ebx, [ebp - 4]
cmp ebx, 0
je Encrypt
cmp ebx, 1
je Decrypt
jmp Failed_Code

Encrypt:
; Push args to the stack
mov eax, [ebp - 8]
push eax
mov eax, [ebp - 12]
push eax

; Call the encryption tool
call encrypt
add esp, 8

; Call printf to give you the encrypted value
push eax
push enc
call printf
add esp, 8
jmp Main_Done

Decrypt:
```

```
; Push args to the stack
mov eax, [ebp - 8]
push eax
mov eax, [ebp - 12]
push eax

; Call the decryption tool
call decrypt
add esp, 8

; Call printf to give you the decrypted value
push eax
push dec
call printf
add esp, 8
jmp Main_Done

; Run here if there was an issue with the arguments
Failed_Code:
push failure
call printf
add esp, 4

; Clean up stack
Main_Done:
add esp, 12
mov eax, 0
ret
```

$$R_{s_1}(b) = (2^{-3})b \oplus b \tag{1}$$
$$R_{s_2}(b) = (2^{5})R_{s_1}(b) \oplus R_{s_1}(b) \tag{2}$$
$$R_{s_3}(b) = (2^{-2})R_{s_2}(b) \oplus R_{s_2}(b) \tag{3}$$
$$R(b) = R_{s_3}(b) \tag{4}$$

$$K_f(k) = \sum_{i=0}^{3} 2^{8i} R\left(\frac{k}{2^{8i}} \bmod 256\right) \tag{5}$$
$$E(v, k) = v \oplus K_f(k) \tag{6}$$
$$D(v', k) = v' \oplus K_f(k) \tag{7}$$

Equations 1 through 4 are performed in a $n \bmod 256$ space (a single byte). Essentially, what is going on in the assembly can be boiled down to these equations. Equations 1 through 4 represent what is happening in the rand_gen subroutine. This is what is known as a Xorshift algorithm, a very simple pseudo random number generator. This will produce consistent results given a seed, and the seed comes from the key you provide to the functions in equations 6 and 7. The encrypt subroutine uses a provided seed to encrypt the key, and the decrypt subroutine does the opposite. The basic idea of this encryption is to generate a number from the key using the Xorshift algorithm, often referred to as hashing. This hash is then used to xor with the original value, which produces our encrypted eax value. These selected functions shown above perform the entirety of the encryption process, while a main function takes care of calling these in order and printing these to the screen. The executable will take command line arguments for the passkey and value and encrypt or decrypt them. To compile the executable, we run ”nasm -f elf32 -g -F dwarf encrypter.s -o encrypter.o” as our assembler and ”gcc -m32 -o encrypter encrypter.o -lc” as the linker. GCC was used to link printf and atoi (used to parse the command line arguments).

An example of the encryption binary in action as a command line tool is shown below.

```
[bentonguess@benton-pc hw1]$ ./encrypter
Arguments: [0 = encrypt| 1 = decrypt] [key] [value]
[bentonguess@benton-pc hw1]$ ./encrypter 0 32288061 3000
Encrypted Value 10407713
[bentonguess@benton-pc hw1]$ ./encrypter 1 32288061 10407713
Decrypted Value 3000
```