CS 478/513: Computer Security
Spring 2020
Programming Assignment 1
Due: Tue., 3/17/20, 11:59 pm

Overview Each part of this assignment is worth **50** points, for a total of **100** possible points. All code must be written modularly in C or C++, and must successfully compile and run on the Linux machines in the CS department's labs.

Part 1: (50 points) Cracking a substitution cipher

This problem is equivalent to Problems 11 - 13 from Chapter 2 of the textbook. However, you need not follow the exact algorithm proposed for Problem 13. You may improve it if you choose to do so.

The provided file `ciphertext.txt` contains a message which has been encrypted using a simple substitution cipher. You are to create a program which can assist a cryptanalyst by providing the following functionality:

1. Compute and display the frequency analysis of the ciphertext.
2. Decrypt the ciphertext using a key provided by the analyst.
3. Guess the key of the cipher.

In order to guess the key, the following procedure should be followed:

1. Compute the frequency ordering of the letters appearing in the ciphertext, and guess a potential decryption key by mapping them to the standard English letter frequency ordering: **ETAOINSHRDLCUMWFGYPBVKJXQZ**.
2. Analyze the quality of your potential key by counting the number of words from the provided `dictionary.txt` file which appear in the decoded message.
3. Permute the guessed key in order to increase the number of dictionary words the decrypted text contains, until further permutations no longer improve its quality.

Your program should print out the final key that it computed as well as the resulting decoded message. Do not forget that your program must also be able to facilitate manual cryptanalysis as described above – your program is not expected to reproduce the exact plaintext, but rather get close enough such that an analyst (or you, in this case) can easily determine the true key.

Hints and Suggestions:

- The textbook suggests shifting letters from the frequency distribution by one position in order to improve the guessed key. You can also try shifting letters by more than one position if it yields a better result. In fact, it may possible to decode the given ciphertext completely, without human intervention, by applying this suggestion.
- Instead of scoring your key by only the number of dictionary words it produces in the deciphered text, you can also try to factor the length of detected words into the score.

Part 2: (50 points) Exploiting a buffer overflow

A buffer overflow is a very serious vulnerability, which has the potential to give a hacker complete control over a system. In this exercise, you will study what makes this a feasible attack vector, and perform a proof-of-concept exploitation of a buffer overflow in a trivial program. The provided `bof.c` contains three functions, in addition to the `main` function:

- `validate_user`, which checks whether an entered password is correct.

- `success`, which is called when the password is correct.
- `failure`, which is called when the password is incorrect.

The `validate_user` function checks the entered password against a constant string which is specified in the code. However, assume you do not know this password – your goal is to gain access by entering a *different* password. Specifically, you must craft an input which will overrun bounds of the `char buff[5]` array in order to manipulate other values on the stack. Your input should cause the CPU to skip over the conditional branch instruction which checks the return value of `validate_user` in `main`, and instead skip directly to the instruction which calls the success function.

You will likely need a hexadecimal editor in order to craft your input. The console-based editor `vi` editor can be used to edit hexadecimal code[1], however GUI editors (e.g. GHex[2], which is available in the labs) may be easier to use.

Buffer overflow is a very common attack vector, so compilers often insert special markers in the stack called stack canaries. The corruption of these canaries will cause the program to terminate. In this scenario, you will need to disable stack canaries — you should use the following command to compile the C code: `gcc -fno-stack-protector -g bof.c`.

To facilitate your attack, you will need to make use of `gdb` and `objdump` to determine relevant memory addresses, their values, and the values of the `rip`, `rbp`, and `rsp` registers. The purpose of `objdump` is to disassemble the compiled binary into assembly, and label each instruction with its memory address. For example, the command `objdump -d a.out > a.dump` will disassemble `a.out` and write the output to the file `a.dump`. You will need the information provided by `objdump` in order to determine the instruction address you want to write into memory.

The GNU Debugger is a very powerful debugging tool which you can use to add breakpoints to your code as well as examine the values of arbitrary registers and memory locations. You will need to use gdb to determine the location of `buff` as well as the values of the memory addresses in the stack. Some useful commands for `gdb` are given in Table 1.

Table 1: `gdb` Commands

Command	Description
i r $xyz	Examine the register **xyz**. E.g., **i r $rsp** to view the value of the **rsp** register.
i ad xyz	Print the memory address of the symbol **xyz**. E.g., **i ad buff** to view the location of **buff**.
x/nbx addr	Print **n** bytes in hexadecimal, starting at address **addr**. Complex expressions are allowed in place of an address, e.g. **x/32bx**(**$ebp − 16**) to print 32 bytes, starting at the location 16 bytes before the value of the **ebp** register.

You will likely need to research the structure of the stack frame used in the x86 architecture, as well as the purposes of the **rip**, **rsp**, and **rbp** registers. You need to write a report which describes how the stack is manipulated when a function is called, how buffer overflows are exploited, and how to fix the vulnerability in **bof.c**. You also need to describe in detail how you exploited the vulnerability, including a list of **gdb** commands you used. The full requirements of the report are explained below.

Submission requirements: Upload a tarball (.tar or .tar.gz) to Canvas, containing two directories – one for each part of the assignment. The submission should include the following material:

Part 1:

1. Your C/C++ code, with a Makefile to compile it.
2. A text file (Readme) explaining how to use your program for each of the following tasks:

[1] http://www.kevssite.com/2009/04/21/using-vi-as-a-hex-editor/
[2] https://wiki.gnome.org/Apps/Ghex

(a) Display the frequency analysis of the ciphertext.

(b) Use the program to guess the cipher key and decrypt the text using it.

(c) Decrypt the text using a key entered by the user.

Your Readme should also explain the algorithm you used to guess the key, if it is different from the one proposed in the textbook.

3. A text file containing the decryption key you obtained and the plaintext it produced. If manual cryptanalysis was required after running your program, you should also explain the steps you took to finish decrypting the message.

Part 2:

1. A copy of the decompiled (objdump) assembly code from the **bof.c** program.
2. Your report (PDF) containing the following information:
   (a) An explanation of what happens in the stack when a function is called, including a description of purposes of the **rbp**, **rsp**, and **rip** registers as used in the Intel x86 architecture.

   (b) A short description of what (in general) happens during a buffer overflow attack.

   (c) An explanation of why the code used to take input in **bof.c** is dangerous, and what procedure should be used instead.

   (d) A list of commands you used in GDB in order to examine the stack, as well as their output at each step.

   (e) A copy of the input you provided to the program in order to perform the exploit.