Name: ____Madina Martazanova____

Computer and Network Security

Lab 4 Buffer Over Flow

For this lab, you need to use a computer. I assume you have access to a Microsoft Windows based personal computer. You also need to install software on C drive under Program Files and hence you need access to a computer with **administrative privileges**.

Please do not apply the techniques you learn in this lab on systems for which you are not responsible and you do not have explicit permission.

0. Set up VirtualBox and the Linux Environment

a. If you have not done so already, follow Lab 1 to set up VirtualBox.

b. Assess **`http://www.itss.brockport.edu/~nyu/security`** and download the **Buffer Overflow** virtual machine file. The file is about 1.35 GB in size. So be patient as the file is downloaded. It will download a file named **`Buf-Over-vm.ova`**.

c. To import the virtual machine, select File → Import Appliance and select the downloaded **`ova`** file. Click **`next`** and **`import`**. This will import a virtual disk image **`Buf-Over-vm-disk1.vmdk`** and thus create the virtual machine. Select the virtual machine **`Buf-Over-vm`** created and see its properties. You will notice that we have installed a 32-bit Ubuntu image.

d. Select the newly created virtual machine and start it. You will see Ubuntu boot up. Login as **`seed`** with the password **`dees`**.

Where you able to successfully import the virtual machine? **_Yes__**
If not, please see me in person and seek help.

If you want to shut down the machine, use the power button on the top, next to the login name **`seed`**. Files created and saved properly will be available the next time you login.

1. Explore the System and the Project Files

Open a terminal window. Perform **`pwd`** and **`ls -l .`** commands to see the contents of the directory you are in (**cd Project**) and navigate yourself to the folder named **`Project`**. Perform again, the command: l**`s -l .`** In this folder, you should see one executable program file.

Name of the executable program file: **__wisdom____**

Name: ____Madina Martazanova____

Run the program **`wisdom`** by typing **`./wisdom`**

When we do this, we see a greeting and listing of four options. As indicated, typing the number **`1`** allows you to "receive wisdom", typing **`2`** allows you to "add wisdom", and typing allows you to “remove wisdom”. Option 4 allows you to terminate the program. Let us try these options one at a time to see if the program works as expected:

Try option **`1`**.

Response received**:** **Enter your selection (1-4)> 1**
Sorry, no wisdom to display!

Try option **`2`**, and add a wisdom: **`8 hours of sleep per day is important`**

Try option **`2`**, again and add another wisdom: **`30 minutes of exercise per day is beneficial`**

Try option **`1`**. What did you observe?

8 hours of sleep per day is important
30 minutes of exercise per day is beneficial

We can keep doing this as long as we like.

Try option **`3`**. What do you observe?

__ This option has not been implemented. ___

Try option **`4`**. What do you observe?

__Program terminated__

When you see the above message, press enter to receive the shell prompt.

2. Bash Script and Entering Non-ASCII Information

To exploit the program later, we may have to enter non-printable characters, i.e., binary data in hex format. To input binary data to the program, we have actually constructed a Bash script to process the input and pipe it to the real executable that is saved as a hidden file (name begins with a period):

Display the content of the **`wisdom`** file using **`cat`** and write the code below.

___ #!/bin/bash
while read -r line; do echo -e $line; done | ./.wisdom __

Name: ____Madina Martazanova____

See the **–e** option for the **echo** command. This allows entering binary-format strings (e.g., with hex escaping).

Perform **man** command for **echo** and determine what the option **–e** allows.

__-e enable interpretation of backslash escapes__

To see the hidden files, perform the command: **ls –al .**

List the names of all three files in the directory: ___ **wisdom, .wisdom, .wisdom.c**
_

The executable **.wisdom** was created from the C source file **.wisdom.c** using the following command:

```
gcc –fno-stack-protector -ggdb –m32 .wisdom.c –o .wisdom
```

Since there are no stack protectors, it is possible to overflow the buffer and perform the exploit. Also note that we are compiling with –m32 option. This ensures that we are creating code for 32-bit architecture.

The Bash script reads user input and pipes it to the executable **.wisdom**. As a result, when **.wisdom** aborts or terminates for one reason or another, we will see a blank line. Press enter once for the Bash script to recognize that **.wisdom** has stopped execution. It will terminate and then a shell prompt will appear. Always run the program using this Bash script.

Let us test the program little bit more. Even though, the system says, we should enter our option in the range 1-4, let us try other options as well to see how well the program holds up. Run **./wisdom**, try each of the following options, and record what you observe:

Option **-10**:

```
Hello there!  What is your wish?
1. Receive wisdom
2. Add wisdom
3. Remove wisdom
4. Quit
Enter your selection (1-4)> 0

./wisdom: line 2:  4385 Broken pipe            while read -r line; do
   echo -e $line;
done
     4386 Segmentation fault      | ./.wisdom
```

Option **0**: __ **Sorry, this option is not valid.___**

Name: ____Madina Martazanova____

Option `2` and enter `\x41\x42\x43 of Healthy Living` as the wisdom

Enter option `1` and view the wisdom entered above. What do you see?

__ ABC of Healthy Living__

Explain the connection between `\x41\x42\x43` and what you see: **These are the ASCII code values associated with 'A', 'B' and 'C.'**

Option `5`: **_ Sorry, this option is not valid._**

Option `10`: **_ The program could not return to the options so I had to quit.**

Option `a`: (i.e., enter a character when a number is expected) "**Sorry, this option is not valid." Was printed on the command line.**

Option `@#$%`: "**Sorry, this option is not valid." Was printed on the command line.**

3. Study Source Code File

Study the source code and answer the following questions.

Like most C programs, it starts with a number of `include` for `header` files. It is followed by type definitions, the definition of a bunch of constant strings for the messages we need to print, and declaration of few external (global) variables. This is followed by several functions.

Write down the names of all functions (including main).

void write_secret(void)

void pat_on_back(void)

void display_wisdom(void)

void add_wisdom(void)

void remove_wisdom(void)

void no_action(void)

int main(int argc, char *argv[])

Name: ____Madina Martazanova____

Observe an array with six components declared and initialized just ahead of the main function.

What is the name of the array? **fptr ptrs[6]**

That array contains the starting addresls lsses of four functions that are invoked if the user enters an appropriate option. Specifically, when user enters option `1`, the program looks up `ptrs[1]`, which is `display_wisdom`, and invokes that function. (See lines 150-153 in the source code.)

Name the four functions: _ **no_action, display_wisdom, add_wisdom, remove_wisdom** _

The main function basically displays the menu allowing four choices, reads user input as a character string into an array named `buf` (carefully ensuring that it reads no more than 1023 characters), converts the input to an integer, uses the integer to index into the `ptrs` array and executes one of the four functions as needed.

There are, however, two functions that do not seem invoked for any specific option in the range 0-5. We would like to find a way to execute these two functions.

The program maintains a linked list of entries provided as wisdom.

`display_wisdom` function basically traverses through the linked list and displays the previously entered wisdom entries. If the list is empty, displays a message to that effect.

`add_wisdom` function prompts the user to user a wisdom and reads the user input into an array. If the user has entered one or more characters, the entered wisdom is added to the linked list.

With the above description and your study of the C code, answer the following:

Thus, there are **three** arrays in the program that can be accessed with index out of bound, leading to buffer overflow.

Name the three arrays:
ptrs[6]
char wis[DATA_SIZE] = {0};

__ buf[r] = '\0';

Two of these array takes user input directly. Thus, with sufficiently long user input these buffers may overflow and can lead to buffer overflow attack. Name the two arrays and identify the specific line code in which they receive user input:

- Name of array: _ **wis[DATA_SIZE]**_
  Line of code in which it receives user input: **/*read user input*/**

Name: ____Madina Martazanova____

r = read(infd, wis, sizeof(wis)-1); /*leave one component for \0*/

- Name of array: **__ buf[1024] __**
  Line of code in which it receives user input: **/*read user selection*/**
  r = read(infd, buf, sizeof(buf)-1); /*leave one component for \0*/

For the third array, we do not receive user input into it directly. But based on the user input, we step out of bounds in the array, and treat the value in that location as the address of the function to be invoked. Name the array and the specific line of code that may cause is to access a memory location out of bounds.

- Name of array: **_ ptrs[6]**

  Line of code in which out of bound access can occur:
  _/*perform user requested action*/
  fptr tmp = ptrs[s];

4. Use Debugger

To exploit the program, you will have to learn some information about how it is laid out in memory. You can find out this information using the **`gdb`** program debugger. You can attach **`gdb`** to your running program, and then use it to print information about the state of that program, and step through executions of that program.

Open two terminal windows side by side. Resize both windows so that you can see both windows fully and can switch back and forth between the two windows. In both windows, you should be in the **`Project`** directory. Run the program in one and run the debugger in the other. Follow the instructions as stated carefully, so you can determine correct memory addresses.

In the first window, run the program by typing: **`./wisdom`**

In the second window, attach **`gdb`** by typing: **`` gdb -p `pgrep .wisdom` ``**

Be sure to use back quotes and not forward quotes. Also note that it is **`.wisdom`** (the hidden executable).

Once you have connected to the process, you can start using **`gdb`** commands to start examining its state and controlling it.

At this point, the execution of that program is paused, and we can start entering **`gdb`** commands.

Name: ____Madina Martazanova____

Place a break point in line **76** of **.wisdom.c** (the source code) by typing the debugger command: **break .wisdom.c:76**

Debugger command used:
break .wisdom.c:76
continue
print &wis

Code for Line 76: **r = read(infd, wis, sizeof(wis)-1); /*leave one component for \0*/**

Place another break point in line **140** of the source code

Debugger command used: __ **break .wisdom.c:140**
continue

_

Code for Line 140: _ **r = read(infd, buf, sizeof(buf)-1); /*leave one component for \0*/**

Enter **continue** in the debugger window to allow the program to continue execution.

This will allow the program to reach line **140**, and pause.

Enter appropriate debugger commands to determine the answers for the following questions. Addresses printed will be in hexadecimal format (a **0x** followed by <u>eight</u> hex digits. If you see fewer than 8 hex digits, add 0's in front). Below are some <u>examples</u> of commands you may need.

To determine the value of a variable **s**, type	**print s**
To determine the address of the variable **s**, type	**print &s**
To determine the address of the function **main**, type	**print &main**
To print the contents of the register **eip**, type	**print/x $eip**
To print in hex, the contents of 32 4-byte words starting from address **bfbbf0c0**, type	**x/32wx 0xbfbbf0c0**

What is the address of **buf** (the local variable in the main function)?

Debugger command used: ___ **print &buf**__
Address determined: __**0xbffff140**__
What is the address of **ptrs**, the external (global) variable?

Debugger command used: __ **print &ptrs** ____

Address determined: _**0x804a034**__

Name: ____Madina Martazanova____

What is the address of the `write_secret` function?

Debugger command used: ___ **print &write_secret___**
Address determined: **___0x80484f4_____**

What is the address of the `pat_on_back` function?

Debugger command used: ____ **print &pat_on_back___**
Address determined: **__0x8048519__**

What is the address of `p`, the local variable in the main function?

Debugger command used: **__ print &p ___**

Address determined: **_0x804a030_**

What is the value contained in the register `ebp`?

Debugger command used: **_ print/x $ebp__**

Value determined: **__0xbffff558__**
At this stage, allow the program to run, by typing the debugger command: `continue`

In the first window, in which the program is being run, enter the option `2`.

This will allow the program to reach line `76`, and pause. Move to the debugger window.

What is the address of `wis` (the local variable in the `add_wisdom` function)

Debugger command used: **_ print &wis__**
Address determined: **__: 0xbffff0c8**

What is the value contained in the register `ebp`?

Debugger command used: **__ print/x $ebp__**

Value determined: **__0xbffff118___**

At this stage, we identified all addresses of interest. Type: `quit` to stop the debugger.

In the first window enter some string as wisdom and quit the program as the wisdom entered is not of any interest.

5. Exploit

Name: ____Madina Martazanova____

With the information you have gathered, compute the necessary values and perform the following exploits.

a. When the program is prompting for an option in the range 1-4, we wish to enter a different number that will execute the `pat_on_back` function. What input number will you provide to the program so that we index out of bounds in the `ptrs` array to access the contents of the variable `p`? You can determine the answer by performing a little arithmetic on the addresses you have already gathered. If successful, you will end up executing the `pat_on_back` function. Determine the smallest integer in decimal.

Microsoft Windows operating system provides a calculator that can be used to do the computation. Change the mode to Programmer. Also remember that we are working on a 32 bit architecture where each address (pointer) is 32 bit or 4 byte long.

Value to be entered: __**-1**__

Run the program and try this input value.

Did you succeed in executing `pat_on_back`?

What message was displayed? ___**Congratulations!** __

How did you compute the number? Show work.

(&p - &ptr) / 4 = (0x804a030 - 0x804a034) / 4 = FFFFFFFFFFFFFFFF = -1

b. When the program is prompting for an option in the range 1-4, we wish to enter a different number that will execute the function whose address appears in `buf[16]` through `buf[19]`. What input number will you provide to the program so that we index out of bounds in the `ptrs` array to access the contents of `buf[16]` though `buf[19]`? Determine the smallest integer in decimal.

Value to be entered: _**-9**_

How did you compute the number? Show work:

0x80484f4 - 0x8048519 /4 = FFFFFFFFFFFFFFF7 = -9

c. When the program is prompting for an option in the range 1-4, we wish to enter the following string in which the **\xNN\xNN\xNN\xNN** portion is suitably replaced. Our objective is to get the program to execute the `write_secret` function.

```
771675207\x00AAAAAA\xNN\xNN\xNN\xNN
```

What do you replace **\xNN\xNN\xNN\xNN** within the following input to the program (which due to the overflow in `ptrs` will access the contents of `buf[16]` through

`buf[19]`) and **`write_secret`** function is executed. Remember that Intel processors use little endian byte ordering.

What should replace **`\xNN\xNN\xNN\xNN`**? **_\xf4\x84\x04\x08 _**

Run the program and try this input value.

Did you succeed in executing **`write_secret`**?

What message was displayed? **Secret revealed: Hard work is the key to your success!**

Why there are exactly **`6`** characters **`AAAAAA`** in the middle of the string?

It creates an array of pointers of size 6.

Comment Summary