Name: _____ **Student ID:**_____

You will need:
Kali (VM)
CYSCA2014InABox (VM)
Windows 95 (VM)
A computer with internet access

COS30015 IT Security

Lab 3 (week 3)

In this lab you will experiment with Buffer overflows in the C language.

1. Using Virtual Machine Launcher, start up the COS30015 / **Kali Linux with local network** VM image.

Select Other… if needed
Log in as *user* (user)
COS30015user (password)

Don't need *root*. Now that we know a bit about security models, we're going to implement the principle of least privilege. *user* is a restricted account.

Let's write a C program:

Start the editor thus:

```
pico memtest1a.c
```

Type in the following code:

```
/* memtest1a.c*/
#include <stdio.h>
#define SIZE 8
void test(int*, char*);
int main(){
      int i = 0;
      char buf[SIZE];
      printf("Type in 5-20 chars into the text buffer\n");
      printf("Watch the value of i \n");
      printf("it will be corrupted when you exceed %i chars\n",SIZE);
      printf("Type \"q\" to exit:\n");
      printf("\t| i posn\t| buf start\t| buf end\t| i value\n");
      do {
            test(&i, buf);
            i++;
      }while(buf[0] != 'q');
      return 0;
}
void test(int *j, char* buf){
      scanf("%s",buf);
      printf("OK.\t| %u\t| %u\t| %u\t| %d\n",j, buf, &buf[SIZE],*j);
      return;
}
```

Use
Control+O to write to file (followed by ENTER)
Control+X to exit

Compile thus:

Name: _____ **Student ID:**_____

```
gcc -o memtest1a memtest1a.c
```

When the function is called, ***i*** is passed by reference. The ***test*** function can access this variable.
buf, being an array, is passed by reference, so the memory location is shared between the main function and ***test***.

and run: `./memtest1a`

Where in memory (the address) is ***i***? What is its value?

	Location (posn)	**Comments**
i	*3216844620*	*4-byte integer*
buffer start	*3216844612*	*Start of user input*
buffer end	*3216844620*	*should be '\0'*
buffer size	*8*	
Bytes between buffer start and i		*e.g. 3216844620 - 3216844612= 8* *input of more than 7 bytes should start to corrupt memory (i.e. value of i)*

Each string entered is appended with a NULL characters (a 0 byte). If you type 8 characters, a 9th character (the NULL) will overflow the array and write over adjacent memory. If this memory is in use, it gets corrupted.

Play with the program (type stuff into it) and record your observations here:

Test No (i)	**Text typed into buf**	**Number of characters**	**Behaviour**
0	*a*	*1*	*0*
1	*aa*	*2*	*1*
2	*aaa*	*3*	*2*
3	*aaaa*	*4*	*3*
4	*aaaaa*	*5*	*4*
5	*aaaaaa*	*6*	*5*
6	*aaaaaaa*	*7*	*6*
7	*aaaaaaaa*	*8*	*0*
8	*aaaaaaaaa*	*9*	*97*
9	*aaaaaaaaaa*	*10*	*24929*
10	*aaaaaaaaaaa*	*11*	*6381921*

INTERESTING THINGS START TO HAPPEN HERE

When a buffer fills up, it writes forwards or backwards depending on the CPU and operating system.

Name: _____ **Student ID:**_____

Type in a few strings smaller than 10 characters – note that the integer ***i*** is counting correctly.

What is the distance (in bytes) between ***buffer start*** and ***i***?

n = (&i - buffer start) = 8 bytes

If you try this in RedHat, the memory locations are further apart – about 13 bytes will overflow the LSB of i

Try inputting that many characters (***n***) to overwrite the variable ***i***.
Try it. Does ***i*** change?

Yes. ***i*** has changed to 0

If it crashes, just start it up again and keep typing longer strings. Watch the value of i.

Try inputting n + 1 characters to overwrite the variable ***i***.
Try it. Does ***i*** change?

Yes. ***i*** has changed to <x> //where x is the ascii value of the 9th character

If a string is really long, you will crash the program (Linux will report a Segmentation Fault).
Code can be injected into the window between buffer overflow and segmentation fault, overwriting other parts of the program such as the ***return address*** and the ***EBP***

More background here:
http://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html

2. Try this program:

On Intel architecture, the last name overflows over the first name. It's different on other CPUs. That's why crackers want to know about the hardware.

```
/* memtest2.c*/
#include <stdio.h>
int main() {
      char first[12];
      char last[12];
      printf("Type in your first name: ");
      gets(first);
      printf("Type in your last name: ");
      gets(last);
      printf("Hello %s %s\n", first, last);
      return 0;
}
```

The compiler warning about **gets** will not stop compilation. You can ignore it.

Unlike ***scanf, gets*** lets you type in spaces and other non-printing characters, so is allows an attacker to enter executable code.

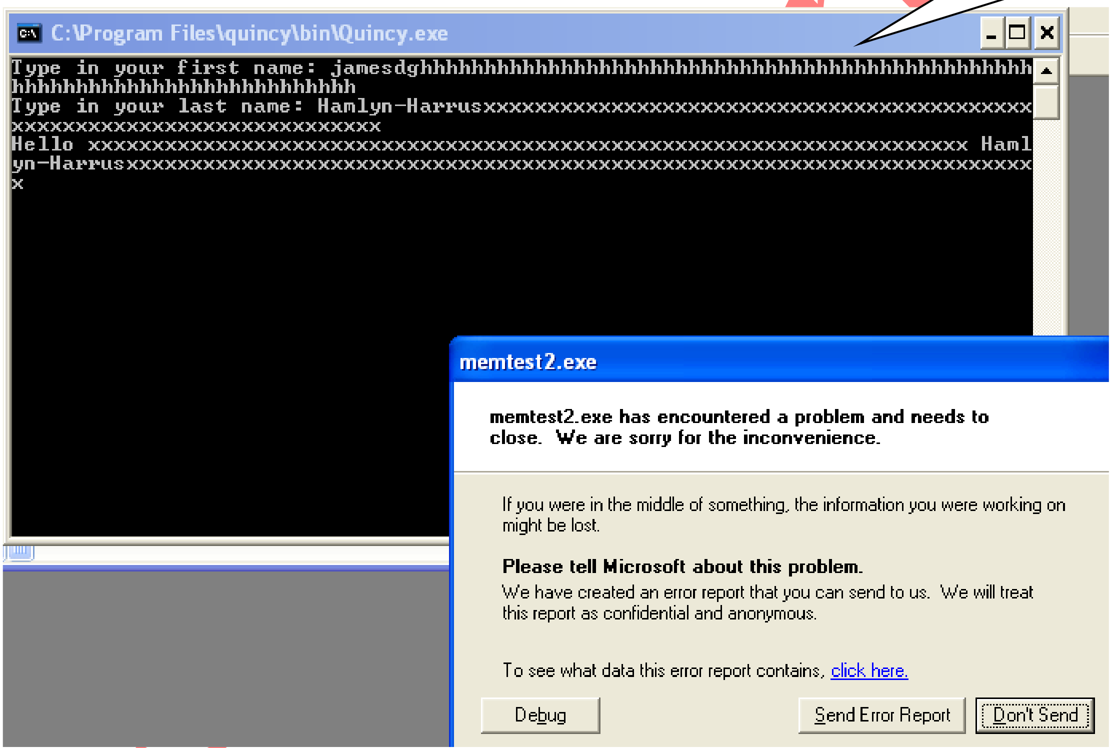
If you put in long strings, you can overflow one string with the contents of the other. If they are big enough, you will get a segmentation fault. If the size is just right, you will overwrite a function somewhere else in the program.

How many characters must you input into the last name to overwrite the first name (try it)?

About 12 or more

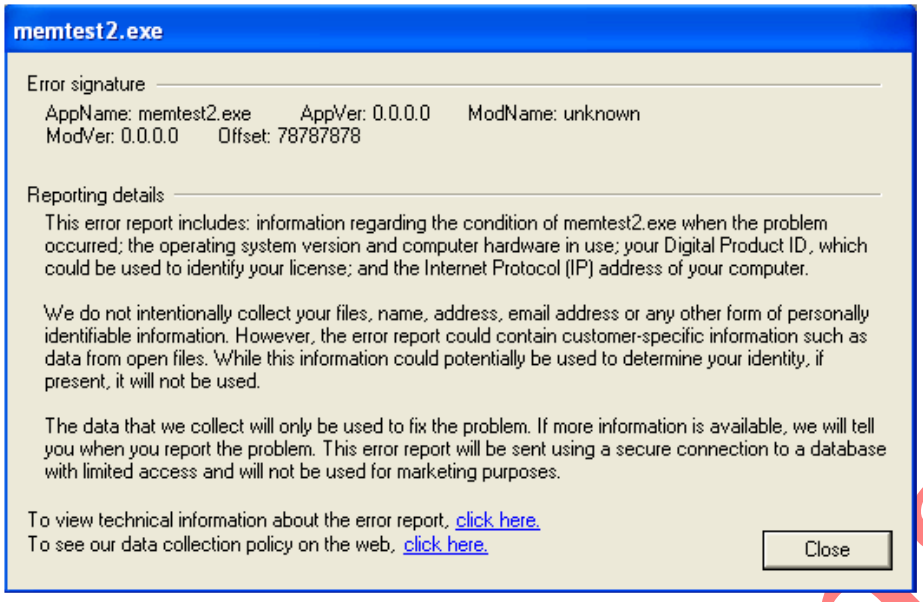
skip this talk if you're running low on time

3. You can also try these programs in Windows using the Quincy IDE (in the Win95 VM). If a 'segmentation error' occurs, Windows will pop up this message:



If you click on click here, you will see the contents of the stack pointer, which contains the hex value of some of the string you typed in.

Name: _____ **Student ID:**_____



Note: 78787878 is hex for xxxx.

Hackers can use this feedback message to figure out which bytes in their exploit string will be copied into the stack pointer. Changing these critical bytes to the location of some (of their) executable code allows the exploit to run their code. With Linux, it's a bit harder, but not much.

skip this talk if you're running low on time

4. Re-write ***memtest1.c*** or ***memtest2.c*** to prevent excess characters being pasted into the char arrays:

HINT:

replace ***scanf("%s")*** with ***scanf("%10s")***

replace ***gets*** with ***fgets***

```
usage: char * fgets ( char * str, int num, FILE * stream );
```

eg. ***fgets(first, 12, stdin)***

Unlike gets, fgets leaves characters in the keyboard buffer.
In windows you can clean up the input stream with

```
fflush(stdin);
```

In Linux you can do the same with

```
char ch;  /*declare once only*/
while ((ch = getchar( )) != '\n' && ch != EOF);
```

example code in ***safegets.c***
on Blackboard.

5. Using Google, look up the various "safe" string manipulating functions available in C. (e.g. strncpy, strncat)
What does the ***n*** do?

Specifies the number of bytes to be processed.

How is value of ***n*** determined?

Set by the programmer.

6. Start up the COS30015 / **CYSCA2014InABox** VM image
In **Kali**, open a web browser (IceMonkey) and surf to `http://192.168.100.210`
Click on CHALLENGES
Scroll down to Exploitation
Read the details of "The Fonz".
Click on the c file (let it open in GVim), and examine the code.

Note that two arrays DestBuffer and FixedVariable are adjacent to eachother. If we can overflow DestBuffer we might be able to write into adjacent variables.

On the host PC, open a browser and go to
https://cyberchallenge.com.au/pdf/CySCA2014_Exploitation.pdf

Follow the Walk-through for "The Fonz". Open a command window and launch netcat:
```
nc 192.168.100.210 20000
```

Don't use the web browser – you can't do interactive input. Netcat (nc) is a universal Telnet-like client suitable for this.

Sending a bunch of **A**'s confirms that the overflow works.

What is the reported contents of FixedVariable?

0x41414141

Connect again and send a series of characters (alphabet) – you can "read" the position of FixedVariable by observing which characters get sent back.

Follow the walkthrough.
```
nc 192.168.100.210 20000
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

What is the reported contents of FixedVariable?

0x54535251

What ASCII characters do these HEX values represent?

T S R Q

HINT-look them up on an ASCII table at asciitable.com on the host PC.

Follow the instructions.

The server wants 0x73696854, which when converted to ASCII and reversed is?

This

Try it out:

```
nc 192.168.100.210 20000
ABCDEFGHIJKLMNOP<your 4 chars answer here>
```

What's the key?

CombatBrownieSwell366

7. On the host PC, surf to http://www.vividmachines.com/shellcode/shellcode.html and read up on injecting shellcode into Linux and Windows processes.

What does . *GetProcAddress()* do?

Returns the memory location of a function in a given DLL library.

8. Using Google, look for tools and programs for checking a C program for potential buffer overflow vulnerabilities and memory leak testers. What are the names of some?

Valgrind, Purify, UMDH, many others.

9. Shut down all VMWare images
and browsers and log off.

End of Lab.