Project 4 Design Document

Overall system design methodology

The system contains three source code files, setup.c, gradebookadd.c, and gradebookdisplay.c, and a makefile. They are compiled into three executables and can be called by command lines. All structs, functions, and variables are defined separately in each source code file.

Data storage

The gradebook is implemented using linked list. A gradebook struct is shown as follow:

```
typedef struct _Gradebook {
  char name[20];
  Assignment *assignments;
  Student *stu;
  int num_as;
  int num_stu;
} Gradebook;
```

A gradebook struct consists of five fields. Its name, two linked lists, which are assignment list and student list, and number of assignment and number of student that keep track with how may assignments and students that have been stored in this gradebook.

Assignments are stored as a linked list. An assignment struct is shown as follow:

```
typedef struct _Assignment {
  char name[20];
  int point;
  float weight;
  Assignment *next_as;
  Student *stu;
} Assignment;
```

It consists of five fields. Three of them are the properties of the assignment: name, point, and weight. It also stores a pointer to the next assignment and a student linked list.

Students are stored as linked lists. A student struct is shown as follow:

```
typedef struct _Student {
  char fn[20];
  char ln[20];
  int grade;
  Student *next_stu;
} Student;
```

It consists of four fields. Three of them are the properties of the student: first name, last name, and grade. It also stores a pointer to the next student.

Grades are stored in student structs. Therefore, student linked lists are not shared among assignments and the gradebook. Each time a student is added, a student struct is created,

initialized, and appended to the end of the student linked list in the gradebook. Then, for each assignment, a deep copy of that student struct is created and appended to the end of its student list. Each time an assignment is added, a deep copy of the student list stored in the gradebook with grade initialized to 0 is created and used by the new assignment. Therefore, all student lists stored in the gradebook and assignments should have the same length and student names at all time.

System IO

A gradebook is stored as a struct in the system and an encrypted binary file outside the system. The setup operation doesn't read in a gradebook. When it is called, it initializes a new gradebook struct, writes the struct into a binary file, and encrypts file. For the other two system operations(add and display), an encrypted file is read in and gets decrypted into a binary file. Then the system reads it to reconstruct the gradebook struct stored in that file. The system carries out the operation that is specified by the command line and prints out error messages if any. When an operation is done, the gradebook struct is written into a binary file and get encrypted.

To read and write a struct that contains linked lists from and to an unencrypted binary file, we implemented six functions. All nodes of the linked lists are written to the file in order after set the pointers in the nodes to NULL. This is because all pointers point to memory addresses. When they are written to a file, memory addresses are no longer there. When reading from a file, all linked lists are reconstructed and the pointers in the node are restored based on the read-in order and set to point to new memory addresses. To decrypt and encrypt a binary file that stores a gradebook, we also implemented two functions encrypt() and decrypt(), the detail of which is introduced in the next section.

```
/*IO functions*/
void store_student(FILE *fp, Student *s) {…
}

void store_assignment(FILE *fp, Assignment *a) {…
}

void store_gradebook(Gradebook *g) {…
}

Student* read_student(FILE *fp,int num) {…
}
Assignment* read_assignment(FILE* fp, int num_stu) {…
}

Gradebook* read_gradebook(char *name) {…
}
```

Linked list operations

The linked list operations in gradebookadd.c and gradebookdisplay.c include adding a node, deleting a node, modifying a node, and accessing a node. All these operations are implemented in the same as a standard linked list project (e.g. CMSC 216 projects).

System stdout

The system will print error/warning message to stdout to notify the user what went wrong.

When a student is added but not any of his/her grades, no grade will be print out for the student if gradebookdisplay -PA or -PS is called for the student. But if -PF is called, all ungraded assignments will be treated as 0 for the student to calculate the student's final grade.

When multiple students with the same name but different upper and lower cases are added in and gradebookdisplay -A is called, it will print the students in the reverse order of the students being added.

Security Analysis

Security Model

To guarantee the privacy and integrity, the gradebook system applied AES-GCM encryption scheme. Each time the encryption function is called, it outputs the ciphertext and a tag(MAC) for authentication. The IV and the tag will then be store in the ciphertext file. To ensure privacy, a new IV is generated and used every time an operation is carried out by the system. It also checks the key. If the key is invalid or of an incorrect length, the system will inform the user that an invalid key is used and exit. To ensure integrity, each time the decryption function is called, it and uses the tag to verify the given ciphertext. If the tag itself or the ciphertext is modified, the decryption process will inform the user that the data has been compromised and exit.

Vulnerability Analysis and Defense

strcpy() vulnerability and Buffer overflow attack
To write a gradebook that contains multiple struct and linked lists into a file, we choose to fix the length of strings (i.e. char* or char[]) to be a reasonable large value. With the use of strcpy(), this implementation brings buffer overflow vulnerability. To counter that, we call strlen() to check input strings. The system will warning the user that a buffer overflow may occur and exit.

```
if(strlen(fn)>100 || strlen(ln)>100){
printf("Buffer overflow warning! Exiting.....\n");
}else{
    strcpy(s.fn,fn);
    strcpy(s.ln,ln);
    add_grade(test,asg_name,&s,grade);
}
```

Authentication and man in middle attack

A man-in-middle attack is likely to occur when the gradebook is on the server. The attacker might attempt to change some bytes of the ciphertext in order to change their grades. A way to prevent this is to use an authentication scheme to make sure that any change to the gradebook's ciphertext can be detected so that the system will be able to inform the legal users of the gradebook. In our gradebook, GCM, an authentication encryption mode, is used to achieve this goal.

```
if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
        handleErrors();
```

In encryption phase, a tag (MAC) is generated and returned to the caller with ciphertext. This is secure since it follows the encrypt-then-authenticate scheme, which we have learned in class. The tag is then stored in the ciphertext file.

```
int ret = EVP_DecryptFinal_ex(ctx, plaintext + len, &len);
    plaintext_len += len;
    EVP_CIPHER_CTX_free(ctx);
    if(ret>0){
        return plaintext_len;
    }else{
        return -1;
    }
```

In decryption phase, a tag is passed in as an argument and used to verify the file. If either the tag or ciphertext is changed, verification will fail. The function will return -1 to warning the user that the file has been compromised.

Reused IV-Key pair vulnerability and attack

The gradebook system is implemented under a symmetric key encryption scheme. An attacker might attempt to exploit some information from the gradebook ciphertext if IV is reused or it's not well randomized. To counter this kind of attack, after each gradebook operation, a new randomized IV is generated in the same way as the keys to ensure randomness and used to encryption the file, even if the gradebook itself doesn't get modified (this happens when gradebookdisplay is called, when there's not data added or deleted from the gradebook).

```
unsigned char *new_iv = (unsigned char*) malloc(sizeof(unsigned char)*16);
    unsigned char *new_tag = (unsigned char*) malloc(sizeof(unsigned char)*16);
    FILE *random = fopen("/dev/urandom", "r");
    fread(new_iv, sizeof(unsigned char)*16, 1, random);
    fclose(random);
```

atoi() vulnerability and integer overflow attack

To convert string to integer, the system uses atoi() to do so. This might lead to integer overflow attack since the length of input string is unpredictable. To prevent this kind of attack, given that we are in a gradebook setting that we can assume a legit input to be a reasonable length, we add the following if condition code to wherever atoi() is called to defense such attacks:

```
if(asg_name!=NULL&&point_s!=NULL && weight_s !=NULL && strlen(point_s)<6 &&
strlen(weight_s) < 6){
            if(roundf(atof(point_s))==atof(point_s))
```

In this code snippet, we assume that a valid maximum points for an assignment should be less 99999(shorter than 6 digits). The second if() condition checks whether the input is an integer.

In add_grade() function, after checking for the above conditions, the system also checks if the input grade is greater than max possible points given an assignment.

```
if(grade > as->point){
            printf("Grade is greater than max possible points.\n");
        }else{
            i->grade=grade;
        }
```

Fake gradebook attack

In our gradebook implementation, a gradebook instance is identified by its file name. This means no two gradebooks should have the same name. However, when an attacker gains access to the programs, he/she might be able to overwrite a gradebook even without knowing the key. This done by calling setup to create a new gradebook using a different name, and then manual change the file name to be the same as the one the attacker wants to overwrite.

Since file names can be easily modified, to counter this attack, we need to encrypt the gradebook name. We include a field called name[] in the gradebook struct.

```
typedef struct _Gradebook {
  char name[20];
  Assignment *assignments;
  Student *stu;
  int num_as;
  int num_stu;
} Gradebook;
```

For all time, the file name should be consistent with the name field stored in the struct. To verify this, each time when a gradebook operation is called, the system will run the following code snippet to compare the two. If they don't match, the operation is terminated.

```
Gradebook *test = read_gradebook(argv[2]);
    if(strcmp(argv[2],test->name)!=0){
        printf("Gradebook name doesn't match.\n");
        return;
    }
```

To launch the above fake gradebook attack, an attacker must use a different name to run the setup command, and then manually change the file name. But since the gradebook name field is stored in the struct and is encrypted, the attacker is not able to modify it to make it consistent with the file name. Therefore, the attack will fail.