A Secure Forum

Introduction

This project is implemented in `C`, in order to improve my secure design mindsets. The program allows user sign up and login user can create post or upload file to the server.

Building.

This program can be compiled and used on OSX and Linux. It is simple as: `➜ forum git:(master) make`

Running.

As this is a client/server program, you need to run them separately.

For server, run `➜ forum git:(master)./server`.

For client, run `➜ forum git:(master)./client`.

Playing.

Here I introduce a simple yet typical usage of secure forum program.

1. Sign up.

Use command `signup` to register, user name and password can only contain alphabetics and digits, special characters are not allowed.

```
➜  forum git:(master) ✗ ./client
forum:> signup
user name:> applicationsec
password:> applicationsec
Register success!
```

2. Login.

Use command `login` to login into the forum, user name and password can only contain alphabetics and digits, special characters are not allowed.

```
forum:> login
user name:> applicationsec
password:> applicationsec
Login success!
```

User account that does not exist will prompt like this: `forum:> login user name:> doesnotexist password:> doesnotexit User name or password is wrong.`

3. Create Post.

Use command 'post' to create a new post. This requires that you are a login user otherwise it will warn you to login. `forum:> post Please login to post new article.`

For login users, you can post with title and content. `forum:> login user name:> admin password:> admin Login success! forum:> post title:> Application Security content:> Application Security Forum Create post success!`

4. Display Post.

Use command `display` to show all posts, after that you can type in article number to read the content.

```
forum:> display
23: application security
24: Application Security
Article number:> 24
Application Security Forum
forum:>
```

5. Upload Files.

Use command `upload` to upload a file located in current directory. Only `.txt` file is allowed to protect the XSS attacks. `forum:> upload file name:> default.txt Upload file success!`

6. Download Files.

Use command `files` to list all files that are stored on server then type in fie number to download it. `forum:> files 1: test.txt: 2: result.txt: ./upload/result.txt 3: default.txt: ./upload/default.txt file number:> 3 Download file ./download/download.txt success!`

7. Exit

Use command `exit` to exit the forum. ``` forum:> exit

➜ forum git:(master) ✗ ```

Vulnerabilities and Protections

The purpose of this project is to examine my *Application Security* mindsets. So here I will demonstrate what I have done to protect my program.

C vulnerabilities

Most vulnerabilities in C are related to buffer overflows external link and string manipulation. In most cases, this would result in a segmentation fault, but specially crafted malicious input values, adapted to the architecture and environment could yield to arbitrary code execution.

1. Use `fgets()` instead of `gets()`.

The stdio `gets()` function does not check for buffer length and always results in a vulnerability. `printf("password:> "); if (fgets(password, PASSWORD_LEN, stdin) == NULL) { fprintf(stderr, "Failed to get password"); continue; }`

2. `strncpy(), strncat(), strncmp()` instead of `strcpy(), strcat(), strcmp()`.

The `strcpy()` built-in function does not check buffer lengths and may very well overwrite memory zone contiguous to the intended destination. ``` src/server.c: if ((strncmp(action->cmd, CMD_SIGNUP, strlen(CMD_SIGNUP))) == 0) {

src/forum.c: strncpy(action.field1, command, strnlen(command, COMMAND_LEN)); ```

3. String formatting attack.

Another important vulnerability is *string formatting attack* which may cause information leakage, overwriting of memory, … This error can be exploited in any of the following functions: `printf, fprintf, sprintf` and `snprintf`.

To avoid this, most of my program is hard code the information while interacting with users. What's more, at least, I never use any user's input.

```
src/form.c:
fprintf(stderr, "Failed to get user name.\n");
continue;
```

4. Off-by-one errors with magic number.

Off-by-one occurs when programmer misuse or hard code the length. To avoid this, all static array must use `macro` for its length. Hard code or magic number are not allowed for this project.

```
src/forum.c:
#define USER_NAME_LEN   1024

char user_name[USER_NAME_LEN + 1] = {0};

if (fgets(user_name, USER_NAME_LEN, stdin) == NULL) {
    fprintf(stderr, "Failed to get user name.\n");
    continue;
}
user_name[strnlen(user_name, USER_NAME_LEN) - 1] = '\0';
```

SQL Injection

Because I use `SQLite` as the back end database to store information. It is a must to considerate `SQL` injection. SQL injection attacks are very common due to two facts: + The significant prevalence of SQL Injection vulnerabilities. + The attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

To avoid this, my considerations contains: + Escaping all user supplied input. + Use of `prepared statement` queries

1. Validation of user input.

To escape and validate user input before putting it in query, I restrict that user input could only contain alphabetics and digits. Do input validation as the first step during the interaction.

```
/**
 * To avoid SQL injection, input data could only contain digit or alphabetic.
 */
static int input_validation(const char *data) {
    int i = 0;

    if (data == NULL) {
        return FORUM_ERR;
    }

    int length = strnlen(data, COMMAND_LEN);
    for (i = 0; i < length; i++) {
        if (isdigit((int)data[i]) || isalpha((int)data[i]) || data[i] == ' ') {
            continue;
        } else {
            return FORUM_ERR;
        }
    }

    return FORUM_OK;
}
```

And here's a typical usage: `printf("user name:> "); if (fgets(user_name, USER_NAME_LEN, stdin) == NULL) { fprintf(stderr, "Failed to get user name.\n"); continue; } user_name[strlen(user_name) - 1] = '\0'; if (input_validation(user_name) != FORUM_OK) { fprintf(stderr, "Input data could only contain digit or alpha\n"); continue; }`

2. Use of `prepared statement` queries

SQLite provides two ways to protect SQL injection: + `sqlite3_exec` and `sqlite3_mprintf` as an all-in-one interface to pre-compile SQL statement. + Use `%q` instead of `%s` to protect SQL injection.

For formatted string, SQLite provides `sqlite3_mprintf` and `%q` to do escape special characters from a formatted query before interacting with the database.

```
src/fdb.c

#define ADD_ACCOUNT_STMT    "INSERT INTO ACCOUNT (name, password) VALUES ('%q
', '%q');"

sql = sqlite3_mprintf(ADD_ACCOUNT_STMT, name, password);
ret = add_value(DB_NAME, sql);
if (ret != FORUM_OK) {
    fprintf(stderr, "Failed to add user account into database.");
}
sqlite3_free(sql);
```

What's more, `sqlite3_exec` is a one-step query execute interface which is a convenience wrapper around `sqlite3_prepare_v2()`, `sqlite3_step()`, and `sqlite3_finalize()`, that allows an application to run multiple statements of SQL without having to use a lot of C code.

```
src/fdb.c:

/* Execute sql statement */
if ((rc = sqlite3_exec(database, sql_stmt, query_callback, (void *)data, &err
_msg)) != SQLITE_OK) {
    fprintf(stderr, "[%d]: Failed to retrieval an entry: %s\n", rc, err_msg);
    sqlite3_free(err_msg);
    (void)sqlite3_close(database);

    return rc;
}
```

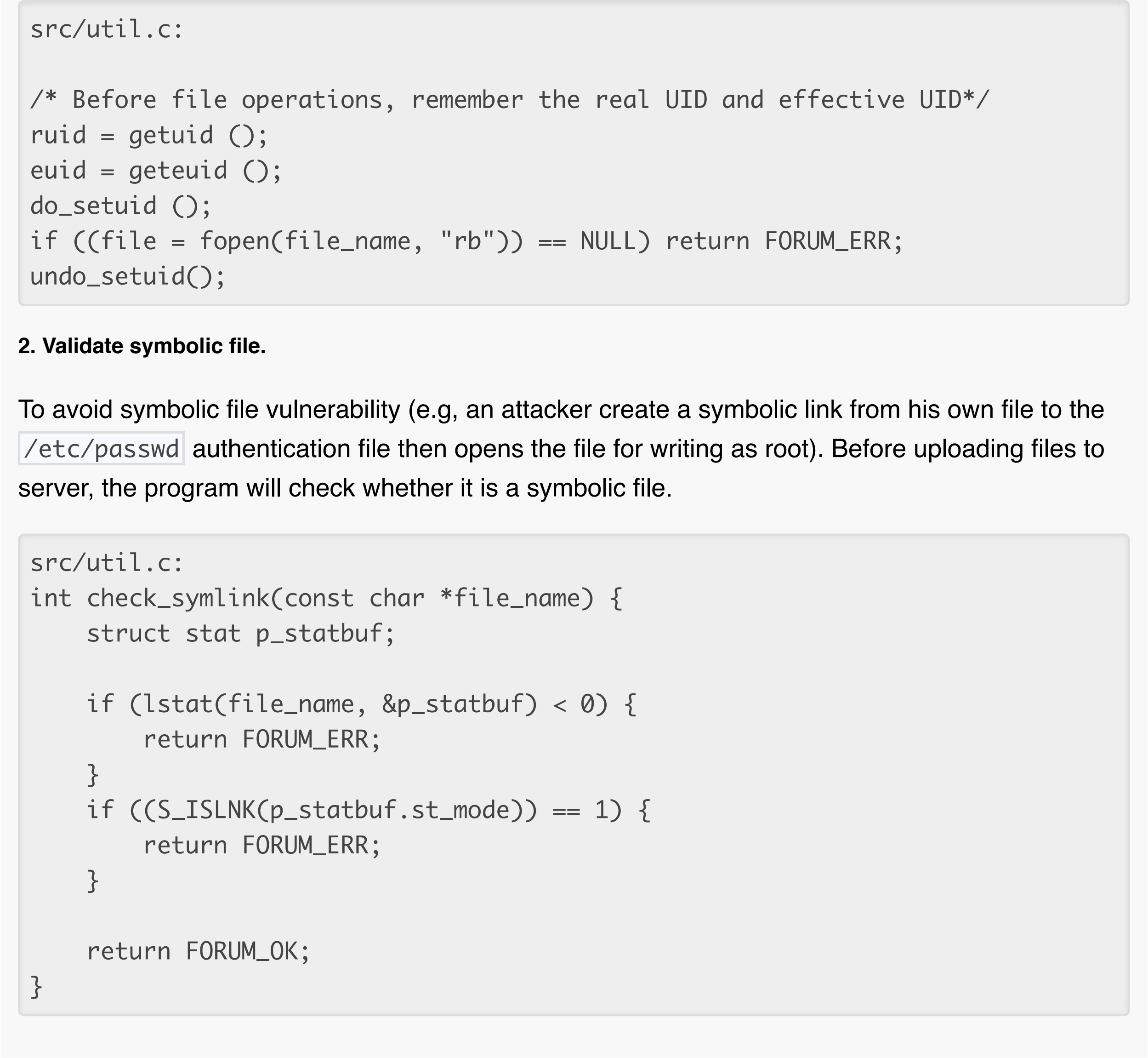
File Vulnerabilities

Considerations of file operations contains: + Execute files by changing the UID. + Execute a symbolic link file which means leak some important information such as `/etc/passwd`

1. `setuid()` to protect privileges

Accessing a file usually indicates a security flaw. If an attacker can change anything along the path between the call to access() and the file's actual use (e.g., by moving files), the attacker can exploit the race condition.

To avoid this, I choose to use `setuid()` to set up the correct permissions while opening file directly and cover it back after accessing a file.



```
src/util.c:

/* Before file operations, remember the real UID and effective UID*/
ruid = getuid ();
euid = geteuid ();
do_setuid ();
if ((file = fopen(file_name, "rb")) == NULL) return FORUM_ERR;
undo_setuid();
```

2. Validate symbolic file.

To avoid symbolic file vulnerability (e.g, an attacker create a symbolic link from his own file to the `/etc/passwd` authentication file then opens the file for writing as root). Before uploading files to server, the program will check whether it is a symbolic file.

```
src/util.c:
int check_symlink(const char *file_name) {
    struct stat p_statbuf;

    if (lstat(file_name, &p_statbuf) < 0) {
        return FORUM_ERR;
    }
    if ((S_ISLNK(p_statbuf.st_mode)) == 1) {
        return FORUM_ERR;
    }

    return FORUM_OK;
}
```