Travail pratique 3

Module - Sécurité logicielle haut niveau

- **Professeur:** Alexandre Duc
- **Assistants:**
  - Robin Müller
  - Axel Vallon
- **Étudiant:** Nelson Jeanrenaud

28 Janvier 2023

Error management

- The code does not manage eventual errors it just crashes the program. This behavior can create issues as it's very easy to crash the program or forces scenarios where the user has no choice but to crash the program. For exemple, if a teacher accidentally select the "See grades of the student", he has no choice to find the name of a student of crash the program by entering one that doesn't exist.
  I removed panic! calls where the error was recoverable by interacting with the user.
- The *show_grades* function contains this line of code

  ```
  println!("The average is {}",
      (grades.iter().sum::<f32>())
      / ((*grades).len() as f32));
  ```

  It is problematic because there is never any verification that the grades vector is not empty. Meaning the program would crash if we ever try to display the grades of a student that has none.

  I added a simple check to see if the collection is empty :

  ```
  if grades.is_empty() {
    println!("No grades yet");
    return;
  }
  ```

Authentification

- The teachers usernames and password are hard coded in the application.

  I'm assuming this was made to simplify the app and to avoid needing a database
- There is no real authentification, there is no way the know who made the modification or accessed sensitive information.

  I added a session variable containing the currently logged in user. This will allow us to later log who made the modification :

  ```
  struct Session {
      username: String,
      is_teacher: bool,
  }
  ```

```
impl Session {
    fn new(username: String, is_teacher: bool) -> Self {
        Self { username, is_teacher }
    }
}
lazy_static! {
  static ref SESSION: Mutex<Option<Session>> = Mutex::new(None);
}
```

- There is no authentification between students except a very scary “Do NOT lie!” message. Any student can see the grades of anyone. Which is obviously a confidentiality issue.

  I added a second HashSet for the student to be able to log in, then I modified the show_grades function to verify that the user is logged in and is either a teacher or a student checking his own grades :

```
fn show_grades_of_student(session: &mut Option<Session>, student: &str) {
    // panic if the user is not a teacher
    match session {
        Some(session) => {
            if session.is_teacher == false && session.username != student {
                panic!("You don't have the right to see the grades of this student");
            }
        }
        None => panic!("You don't have the right to see the grades of this student"),
    }
    println!("Here are the grades of student {}", student);
    let db = GRADE_DATABASE.lock().unwrap();
    match db.get(student) {
        Some(grades) => {
            println!("{:?}", grades);
            println!(
                "The average is {}",
                (grades.iter().sum::<f32>()) / ((*grades).len() as f32)
            );
        }
        None => panic!("User not in system"),
    };
}
```

- The password input of the user need to be obfuscated as he types it in the console.
  I tried using the rpassword crate to not echo the user’s password input but I couldn’t make it work so I didn’t include it in the modifications :

```
fn get_password(message: &str) -> Option<String> {
    println!("{}", message);
    stdout().flush().unwrap();
    let password = rpassword::read_password().unwrap();
    if password.len() > MAXIMUM_PASSWORD_LENGTH {
        return None;
    }
    Some(password)
}
```

Logging

- In the *become_teacher* function, the application logs an error message upon a failed authentification, displaying in clear the user’s (username, password) attempt. This is terrible as it logs a password in clear, the user might have miss typed by only one character his password or username which would reveal his password. All in all there is no valable reason to do this.

  I rewrote the become_teacher function to return a generic message upon login failure :

```
fn become_teacher(teacher: &mut bool) {
    match login() {
        Ok(result) => {
            if result {
                *teacher = true;
            } else {
                *teacher = false;
                println!("Wrong credentials");
            }
        }
        Err(_) => println!("Failed login"),
    }
}
```

- The application should log important actions such as adding or showing grades.

  To do this I added a logger to write the logs in a file, in a real world case I would store them in a separate database from the data.

```
fn main() {
    CombinedLogger::init(vec![
        TermLogger::new(
            LevelFilter::Info,
            Config::default(),
            TerminalMode::Stderr,
            ColorChoice::Auto,
        ),
        WriteLogger::new(
            LevelFilter::Info,
            Config::default(),
            OpenOptions::new()
                .create(true) // to allow creating the file, if it doesn't exist
                .append(true) // to not truncate the file, but instead add to it
                .open("info.log")
                .unwrap(),
        ),
    ]).unwrap();

    info!("{} : Program started", Local::now().format("%Y-%m-%dT%H:%M:%S"));
    welcome();
    loop {
        menu();
    }
}
```

Encryption

The passwords of the users are stored in the database in clear, without any encryption. Granting too much trust into the server and database and risking the user's data in case of database dump or if an attacker access the application memory/code.

As such, I hashed the passwords with argon2 before storing them. The passwords are still in clear in the code but I'm considering this to be out of the scope of this lab since they would not be stored in a hashmap but a database in a real scenario.

```
fn hash_password(password: &str) -> String {
    let salt = SaltString::generate(&mut OsRng);

    let argon2 = Argon2::default();

    let hashed_password = argon2.hash_password(password.as_bytes(), &salt);

    return hashed_password.unwrap().to_string();
}

fn verify_password(password: &str, hash: &str) -> bool {
```

```
    let parsed_hash = PasswordHash::new(&hash).unwrap();
    match Argon2::default().verify_password(password.as_bytes(), &parsed_hash) {
        Ok(_) => true,
        Err(e) => {
            error!("{}", e);
            false
        }
    }
}
```

Input/Output validation

- The input validation in the *enter_grade* method accept grades at any number of decimals, which will be truncated or rounded later which could cause errors.

  > Exemple : 5.55555555555555555555555555555555555 becoming 5.55555553

  To fix this I added checks to the input :

  ```
  let grade: f32 = match input()
          .add_test(|x: &String| x.parse::<f32>().is_ok())
          .add_test(|x: &String| {
              let grade: f32 = x.parse().unwrap();
              grade.trunc() >= 0.0 && grade.trunc() <= 6.0 && (grade * 10.0).fract() == 0.0
              && !(grade.trunc() == 6.0 && grade.fract() != 0.0)
          })
          .try_get() {
          Ok(grade) => {
              let grade: f32 = grade.parse().unwrap();
              grade
          }
          Err(_) => panic!("Invalid grade"),
      };
  ```

- Also the value of the name variable is not sanitized which could lead to injection attacks.

  I used the sanitizer crate to validate the user's input :

  ```
  fn sanitize_name(name: String) -> String {
      let mut sanitize = StringSanitizer::from(name);
      sanitize.trim().alphanumeric().to_lowercase().clamp_max(MAXIMUM_USERNAME_LENGTH).get()
  }

  fn sanitize_password(password: String) -> String {
      let mut sanitize = StringSanitizer::from(password);
      sanitize.trim().clamp_max(MAXIMUM_PASSWORD_LENGTH).get()
  }
  ```

- There is no maximum length for the name and grade which could lead to buffer overflow attacks.

  So I added a test and a constant to block the length of the string :

  ```
  let name: String = input()
          .add_test(|x: &String| !x.is_empty() && x.len() <= MAXIMUM_USERNAME_LENGTH).get();
  ```