Labo 3 - Access control

SLH

Timothée Van Hove

January 22, 2024

Contents

Introduction

The goal of this lab is to detect security vulnerabilities and bad practices in a provided rust console application. In addition, the access control should be managed with an access control tool (ex. casbin) and not manually.

Vulnerabilities found in the given code

Input validation

Virtually no input validation is performed in the prompts of the code provided. The user can enter empty text in any input, or enter a grade > 5 in a review, or entering arbitrary long text in any user input. In addition, there is no verification if the user already exists, leading to the overwrite of the user. I therefore decided to implement an input length check (min-max) in each user prompt to avoid any risk of DoS. All the input validation is done using `inquire` Validators. In addition of the length check, I set custom validators in user prompts where it was needed.

I decided to implement the password validation step with 2 calls to the `Password::new`. I did it this way because if I did it with a password confirmation, the `with_validator` function in the `inquire` crate is invoked for each password entry, including the confirmation. This is a problem when the user enters an invalid password as a confirmation, because the displayed error will be the one returned by the validator, and not the `custom_confirmation_error_message`. This way, the "Password do not match" error in the password confirmation is displayed instead of another error.

```rust
// Checks if the password is valid
let name = username.clone();
let password_validator = move |input: &str|
    match input_validation::validate_password(input, Some(name.as_str())) {
        Ok(()) => Ok(Validation::Valid),
        Err(e) => Ok(Validation::Invalid(e.into()))
    };

// Prompt the user for a password
let password1 = Password::new("Entrez votre mot de passe : ")
    .with_display_mode(PasswordDisplayMode::Masked)
    .with_validator(password_validator)
    .without_confirmation()
    .prompt()
    .map_err(|e| { internal_error!("Error in password1 prompt: {}", e) })
    .unwrap();

// Checks if the passwords match
let passwords_match_validator = move |p2: &str|
    match input_validation::do_passwords_match(password1.as_str(), p2) {
        Ok(()) => Ok(Validation::Valid),
        Err(e) => Ok(Validation::Invalid(e.into()))
    };

// Prompt the user for a password confirmation
let password2 = Password::new("Confirmez votre mot de passe : ")
    .with_display_mode(PasswordDisplayMode::Masked)
    .with_validator(passwords_match_validator)
    .without_confirmation()
    .prompt()
    .map_err(|e| { internal_error!("Error in password confirmation prompt: {}", e) })
    .unwrap();
```

Authentication

No condition were implemented for the password strength or length in the registration process. I decided to use `zxcvbn` to enforce a strong password (also by providing the username) and enforce the length to be between 8 and 64 characters.

Plaintext credential storage

Well.., it's hard to do any worse, so passwords are now stored hashed and salted in the database with argon2. The user password verifying process is done in a time-constant way, to protect against timing attacks.

Error handling

Insufficient error handling

In the given code base, almost no error is handled. This leads the program to panic when something go wrong. For instance, when the credentials are wrong in the login process, or during "Ajouter un avis" and "Supprimer un avis" actions the program just crashes.

In my implementation, I focused on error handling. In the event of an error, the program simply displays "Erreur interne" to the user and the main menu is displayed.

Information leaks

The few prompted errors did leak too much information. For instance, in the login process:

```
let user = User::get(&username).expect("l'utilisateur n'existe pas");

if password == user.password {
    loop_menu(|| user_menu(&user));
} else {
    println!("Le mot de passe est incorrect");
}
```

In this case, we don't want to be too precise about what went wrong in the login process. The error should instead be displayed as :

`"Nom d'utilisateur ou mot de passe incorrect"`

Note: I want to mention that in my new implementation I do leak the information of an already existing user, an already existing establishment in the `register()` function. This is necessary in our case, otherwise the user cannot understand why the register process fails.

Authorization

Because the goal of this lab is to handle access control with casbin, I won't go deep in the access control vulnerabilities in the given program because I have re-implemented it from scratch. The problems I observed were:

- An Owner and a Reviewer can read everyone's reviews.
- An Owner can review his own restaurant
- Anybody can delete any reviews because of this gem in the code:

```
let is_admin = Confirm::new("Êtes-vous administrateur ?")
    .with_default(true)
    .prompt()?;

if !is_admin {
    bail!("vous n'êtes pas administrateur")
}
```

Casbin model

This model handles requests containing a subject (User structure), an object (String) and an action (Action enum). I define the policy as 2 rules to be evaluated. One rule for the subject and one rule for the action.

I've decided to evaluate the action, because I'm using a serialized enum. This allows me to evaluate the action as: `r.act.name == "WriteReview"` in the `policy.csv` file.

I could have simply handled the action in the matchers as `p.act == r.act`, but in this case I would have had to deserialize the action and pass the string through the enforcer each time.

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub_rule, act_rule

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = eval(p.sub_rule) && eval(p.act_rule)
```

Policies

```
p, r.sub.role.name == "Reviewer" || r.sub.role.name == "Owner" || r.sub.role.name ==
 "Admin", r.act.name == "ReadOwnReviews"
p, r.sub.role.name == "Reviewer" || r.sub.role.name == "Admin", r.act.name ==
 WriteReview"

p, r.sub.role.name == "Owner" && r.sub.role.owned_establishment != r.obj, r.act.name ==
 "WriteReview"
p, r.sub.role.name == "Owner" && r.sub.role.owned_establishment == r.obj, r.act.name ==
 "ReadEstablishmentReviews"

p, r.sub.role.name == "Admin", r.act.name == "ReadEstablishmentReviews" || r.act.name ==
 "DeleteReview"
```

Miscellaneous bugs

The DB is only saved when the user decides to quit properly the application. This could be a problem if for some reason the user decides to enter `ctrl + c`, the database would not be saved. Instead of saving the DB when the user quits the program, I decided to save it each time an new element is created (registration + review), or deleted.

UX issues

- The `PasswordDisplayMode` was `Hidden`. This is a problem because `inquire` does not reset the input in case of a validator error, so the user doesn't know that he has to delete the input. I set it to `Masked` to be clearer for the user.
- I didn't want the user to be stuck in a validation loop. For instance, in the login process, in case of an invalid password or username, instead of looping on the prompt, where the user could be stuck if he didn't remember his credentials, I simply display the error and go back in the main menu.
- I added user confirmation for each user action (Register, Login, Add review, delete review). Like that, the user clearly knows what is the result of his actions.

Code best practices

The user, Role and Review are implemented in the main file. I created separate "models" module for a better separation of concerns.

I factored the code to avoid code duplicates as much as possible. For instance, I created a macro that allows to log an error message, display a generic error to the user, and return to the main menu:

```
#[macro_export]
macro_rules! internal_error {
    ($($arg:tt)*) => {{
        use ansi_term::Colour::Red;
        use log::error;

        let formatted_msg = format!($($arg)*);
        error!("{}", formatted_msg);
        println!("{}", Red.paint("Erreur interne"));
        return ShouldContinue::Yes;
    }}
}
```

As another example, I factored the prompt of a simple text, with simple validators that I use many times in the code:

```
fn empty_validator(object: &str) -> MinLengthValidator {
    min_length!(1, format!("{} ne peut pas être vide", object))
}

fn max_length_validator(length: usize, object: &str) -> MaxLengthValidator {
    max_length!(length, format!("{} ne peut pas dépasser {} caractères", object, length))
}

fn prompt_text(object: &str, length: usize) -> InquireResult<String> {
    Text::new(&format!("Entrez {}: ", object))
        .with_validator(empty_validator(object))
        .with_validator(max_length_validator(length, object))
        .prompt()
}
```

Tests

Unit tests

Unit tests have been implemented for the input_validation and hashing modules. The access control policies have also been tested in the enforcer module.

Manual access control tests

I didn't have time to implement integration tests to validate the access control, so instead I did the tests manually:

- Admin
  - ☒ Can read his own reviews
  - ☒ Can read reviews of any establishment
  - ☒ Can write a review for any establishment
  - ☒ Can delete any review
- Owners
  - ☒ Can read his own reviews
  - ☒ Can read reviews of his own establishment
  - ☒ Can not read reviews of other establishments
  - ☒ Can not write reviews for his own establishment
  - ☒ Can write reviews for other establishments
  - ☒ can not delete any review
- Reviewers
  - ☒ Can read his own reviews
  - ☒ Can not read reviews of any establishment
  - ☒ Can write a review for any establishment
  - ☒ Can not delete any review