ReadMe

1. General Information

1.1. Objective

The objective is to implement a Python code, that showcases at minimum one mitigation solution for security threats to an appointment and scheduling management information system (ASMIS) of a hospital. The ASMIS is a multi-user system serving patients as well as hospital and IT staff. Since the ASMIS will be a web based application and since it will contain Personal Health Information (PHI), which is a high value target for cyber criminals, the ASMIS is likely to be attacked in a real world scenario.

1.2. Threats

There are many ways to get to the valuable asset PHI. You for example can simply spoof the identity of another individual. To mitigate this risk a user authentication, which only permits the correct individual access should therefore be implemented. But not only authentication is necessary, you also should take care of authorization, because you simply want to avoid that a person gets hold of information he is not authorized to obtain. To mitigate this authorization checks must be implemented whenever information is requested or changed from the ASMIS. Within the code of this ASMIS version an `auth` module is therefore implemented. This module can take care of

- user authentication (compare `Authenticator` class in section 3.1 for more information)
- user authorization (compare `Authorizer` class in section 3.1 for more information)

To mitigate successful information disclosure attacks, which can be conducted e.g. via SQL injection (SQLi)attacks, another valuable layer of defense was added in the code. This defensive mechanism is mainly used to secure the passwords of all users of the ASMIS.

- salted hash (compare section 3.2. for more information)

To prevent brute force attacks, which can lead to tampering or disclosure of information the following was implemented in this version of the ASMIS:

- password policy (compare section 3.3 for more information)

1.3. Limitations

This versions of the web-based appointment and scheduling management information (ASMIS) system is to show some security functions. The ASMIS is not fully implemented.

- To showcase some functionality only some data is stored at the "backend". To get an idea what data is kept at the backend use the script ***read_out_backend.py***
- If you need to reset the backend data use the script ***factory_reset.py***
- This test version of the ASMIS is not using SQL as backend. It is using Pickle for simplicity. Pickle is however unsecure and should be replaced.

1.4. Dependencies

- `secrets` module (supported & integrated by Python versions >= 3.6) https://pypi.org/project/python-secrets/

- `pickle` module https://docs.python.org/3/library/pickle.html
- `hashlib` module https://docs.python.org/3/library/hashlib.html
- `os` module https://docs.python.org/3/library/os.html

2. How to execute the code

You can use a simple GUI to check the functionality of the ASMIS. To do so simply start the script ***main.py*** To Login you may use the following case sensitive credentials

username	password	comment
tim	qwerty1234	is only authorized to view his health record
steve	yolo007	can also view other patient records
Administrator	password	is authorized to do all actions of this ASMIS version

3. Description of the solution implemented

3.1. auth Module

The auth.py module was created based on information from Phillips (2018) The following code is used for authentication:

```
    def login(self, username, password):
        '''This method logins in a user if the credentials are correct and if
        the user is not already logged in. The method takes username & password
        as arguments. The method raises excpetions in case the credentials are
        not correct or if the user is already logged in. It has no return value
        '''
        if (self.is_logged_in(username)) == True:
            raise UserAlreadyLoggedIn(username)
            ''This prevents, that a user logges in twice''''
        users = []
        '''flush array before restoring from backend'''
        with open(USER_PICKLE, 'rb') as f:
            users = (pickle.load(f)) #load objects from pickle
            '''restore objects from pickle backend'''
        for user in users:
            salted_hashed = self._salt_hash_pw(password, user.salt)
            if user.username == username and salted_hashed == user.password:
            '''check if we have username AND password match'''
                self.logged_in_users.append(username)
                '''record the name of the user that logged in'''
                return
                '''if we have a match exit the function with no exception'''
        raise InvalidCredentials(username)
        '''if there is no match, raise an exception'''
```

The following code is used in the `Authorization` class:

```
    def is_authorized (self, user, task):
        '''This mehtods checks, if a given user authorized to do a certain
```

```
        task. The method takes the name of the task and the name of the user
        as input argument. The method will return True if the user is
        privileged to do the given task. The method will return False, if the
        user is not privilege to do the given task'''
        return_value = False
        acl = []
        '''flush array'''
        with open(ACL_PICKLE, 'rb') as f:
            acl = (pickle.load(f)) #load objects from pickle
            '''restore objects from pcikle backend'''
        for acl_entry in acl:
            if acl_entry.username == user:
                if task in acl_entry.user_permissions:
                '''check if the user is allowed to do the task'''
                    return_value = True
                    '''if so change the return value to True'''
        return return_value
```

it can be called within the main function of the ASMIS like the following

```
authorizer = Authorizor() # start up a authorizer object


if authorizer.is_authorized(username,'read_own_medical_history'):
    read_own_medical_history()
    '''if the authorizer says that person is authorized the function
    can be started'''
else:
    print("You do not have permission to do this task")
    '''if the authorizer says that person is not authorized an error
    message appears'''
```

The `auth.py` module comprises of the following elements:

- `Authenticator` class
    - The attribute `logged_in_users` stores all logged in users. This version of the ASMIS is however not split in client and server side. This array will therefore only ever store one user, since there are no multiple clients. Due to this this array will be used for a purpose other than intended. This array will not be used to identify all logged in users. This array will be used as the "client cookie" here. In a true client server setup this array of course may not be used as the client's cookie
    - `_salt_hash_pw(password,salt)` method is a class internal function. It is used to store passwords securely (see section 3.2 for more information) via pickle. It takes password and salt as argument. It will return the salted hash
    - The method `add_user(username,password)` can add users to the ASMIS. This method utilizes the afore mentioned `_salt_hash_pw` method. The method takes username & password as arguments. The method raises excpetions in case a username is already taken or if the password is to short. It has no return value

    - The method `login(username,password)` logins in a user. The method takes username & password as arguments. The method raises exceptions in case the credentials are not correct or if the user is already logged in. It has no return value
    - The method `is_logged_in(username)` checks if a user is logged in. It will take a username as an argument. It will return either True or False

- `Authorizer` class

    - The attribute `global_permissions` stores the global permissions any user can have. Note: The permissions are not stored in the backend within this test version of the ASMIS
    - The method `add_new_global_permission(new_permission)` creates a new global permission that users can be obtain. This method takes new_permission as an argument. If the new global permission already exists it raises an error. The method has no return argument.
    - The method `change_user_permissions(perm_name,username,action)` can change permissions an individual user does have. It for example can either grant user xyz the privilege to access method zxy or it can revoke the privilege to do so. The method takes the name of the permission you want to change. It takes the action (revoke / provide). And it takes the user name. The function has no return value. The function raises exception in case e.g. user or permission does not exist.
    - The method `print_user_permissions(username)` prints permissions of a given user. The method takes the name of the user as an input. It has no return value. It raises an exception, if the user is not known.
    - The method `is_authorized(user,task)` checks, if a given user authorized to do a certain task. The method takes the name of the task and the name of the user as input argument. The method will return True if the user is privileged to do the given task. The method will return False, if the user is not privilege to do the given task

- `User` class

    - This class has `username`, `password` `medical_history` & `salt` as attributes. Salt is a random string, that is added prior hashing the password in a hopefully unknown way -- keep the source code hidden :P -- to the hacker so the hash is more tricky to break. The arguments are stored via pickle.
    - The method `read_other_patients_medical_history(username)` This method is to view certain patient data. This method can view very sensitive information. Only certain users should be able to view it. This method has username as input. This function raises exceptions (e.g. if the user is not known). This function does not have a return value.
    - The method `read_own_medical_history()` is to view your patient data (if any). Since the method can only view your information, more users may be allowed to use this methods. This method has no input. This method has no return value

- `AccessControlListEntry` class

    - This class is used to store a users permissions. An example would be: User "Tim" is permitted to do "show_patient_history". Attributes are `username` & the array `user_permissions`
    - The class does not have methods.

3.2. Salted hash

A securely randomly generated string is generated with the help of the `secrets` module. This string is typically called salt.

```
import secrets                                          #used to get a secure inital salt
salt = (secrets.token_hex(32))                          #create new salt
```

This salt will then be concatenated, XORed, ... (here added) with the password of a user. This merged string than will be securely hashed (here SHA265). Hashing the password leads to the fact, that the password is never stored in clear text in the ASMIS. Salting prior hashing will protect against pre-computed hash digest of the most common passwords.

```
salted =  salt + password                               #adding salt & password
salted = salted.encode('utf8')                          #convert to utf8 for sha256
hashing
salted_hashed = hashlib.sha256(salted).hexdigest()   #sha256 hahsing it
```

The password is never stored in the clear text at the backend. Only the salted hash is stored. If the password database gets leaked the password can only be reverse engineered with uneconomical effort. To authenticate a user during login, the password provided by the user at login time is again salt hashed and compared with the stored value. If there is a match, access will be granted. If there is not, access will not be granted. The salted hash is based on information from Howard & LeBlanc (2002)

3.3. Password policy

Another simple but powerful mitigation is to have a minimum length for passwords. A brute force attack is considerably slower, if the password has a sufficient length. The below code snippets shows the implementation within this version of the ASMIS.

```
if len(password) < 8:                                  #check if password length is too
short
    raise PasswordTooShort(username)                   #an exception will be raised, it
the password is too short
```

The password policy can easily be extended by e.g. having rules that numbers, letters & special signs should be within the password. This however will not mitigate dictionary attacks.

4. System test

For test data utilized & results obtained, please see the document system_test.pdf

5. References

Phillips, D. (2018) *Python 3 Object-Oriented Programming*. Third Edition. Birmingham: Packt Publishing Howard, M. LeBlanc, D. (2002) *Writing Secure Code*. Second Edition. Redmond: Microsoft Press