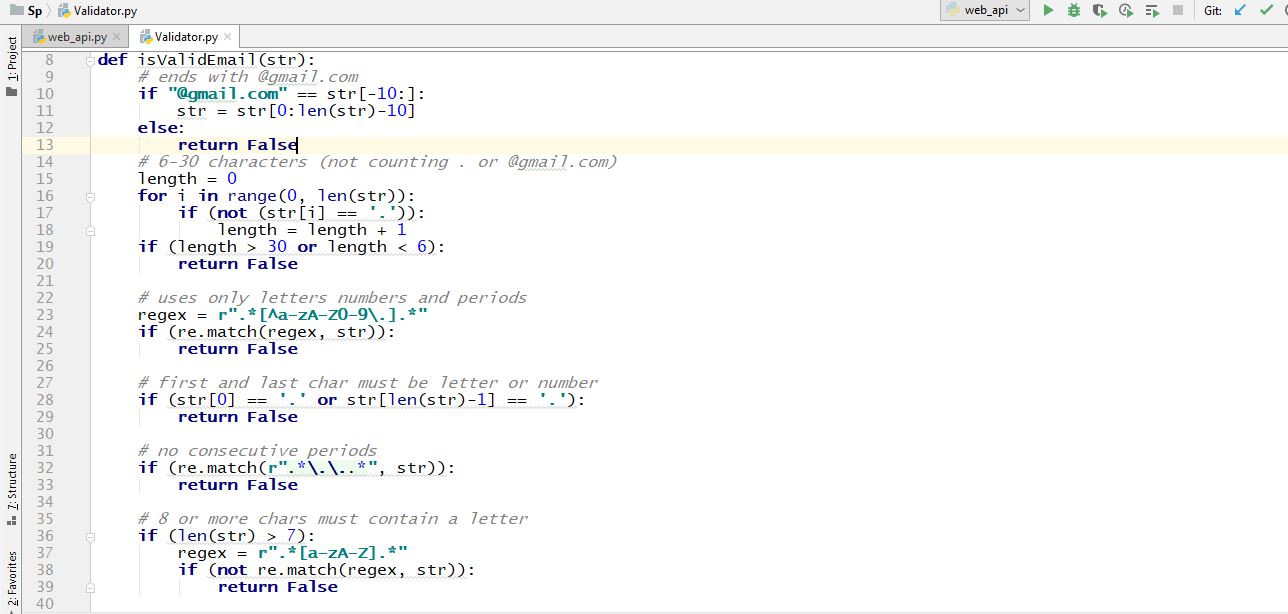
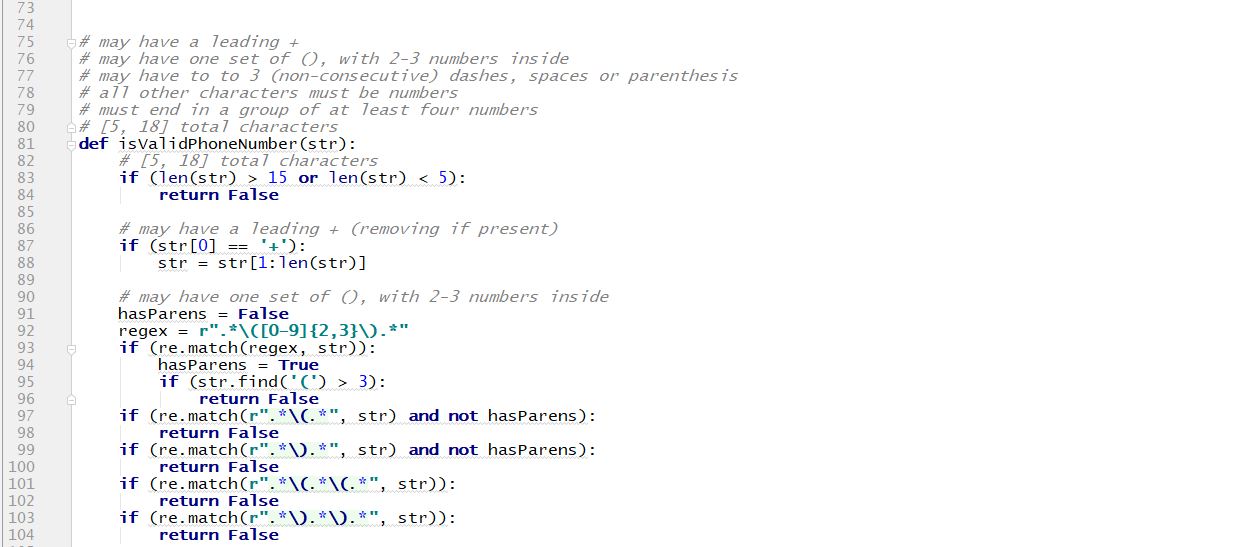
SECURITY CONCERNS AND MITIGATIONS

# Input / Output Validations

## Sample UI Validations



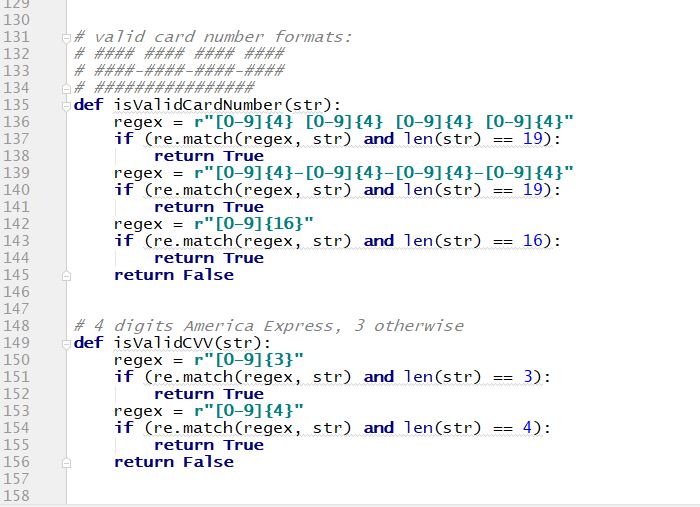


This is the screenshot for the snippet of code Input validation (python file). Here we use regex to verify the   
first name, last name, email id, phone number. We get these values from the users and if check whether they are valid for our system.   
Our Assumptions   
- may have one set of (), with 2-3 numbers inside

* may have to to 3 (non-consecutive) dashes, spaces or parenthesis

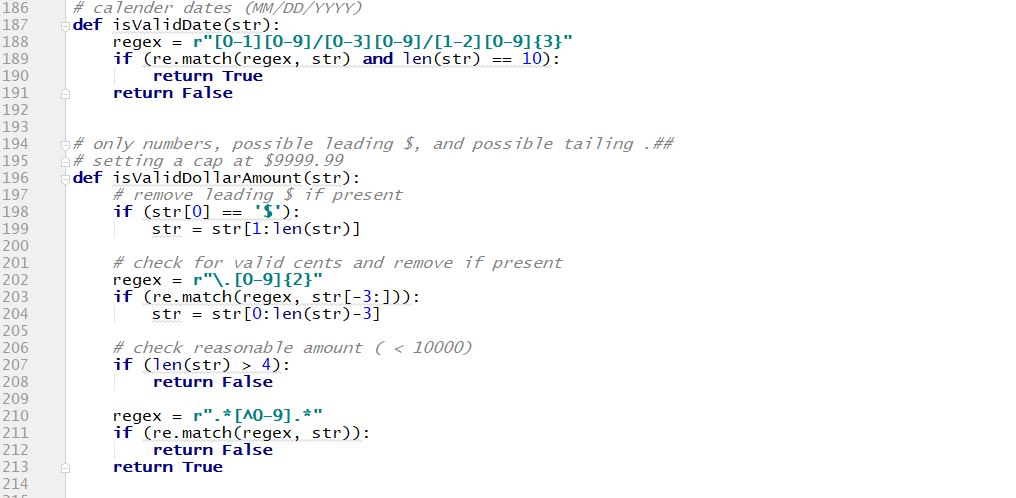
- all other characters must be numbers

- must end in a group of at least four numbers



This is the screenshot for the snippet of code Input validation (python file). Here we validate the credit card number and CVV.   
Our Assumptions are  
valid card number formats:  
#### #### #### ####  
####-####-####-####  
################

3 Digits for normal CC



## Sample Server Validations

|  |
| --- |
| # Assumptions for a valid name (subject to change):  # Only letters, no more than three spaces,  # 0-2 non-consecutive or each hyphen, period, and single-quote  # length has 1-50 letters  def isValidName(str):  if (len(str) > 50 or len(str) < 1):  return False  regex = r".\*[^a-zA-Z0-9\-\.' ].\*"  if (re.match(regex, str)):  return False  if (re.match(r".\*\-\-.\*", str)):  return False  if (re.match(r".\*\.\..\*", str)):  return False  if (re.match(r".\*''.\*", str)):  return False  if (re.match(r".\* .\*", str)):  return False  if (re.match(r".\*\-.\*\-.\*\-.\*", str)):  return False  if (re.match(r".\*\..\*\..\*\..\*", str)):  return False  if (re.match(r".\*'.\*'.\*'.\*", str)):  return False  if (re.match(r".\* .\* .\* .\* .\*", str)):  return False  return True |

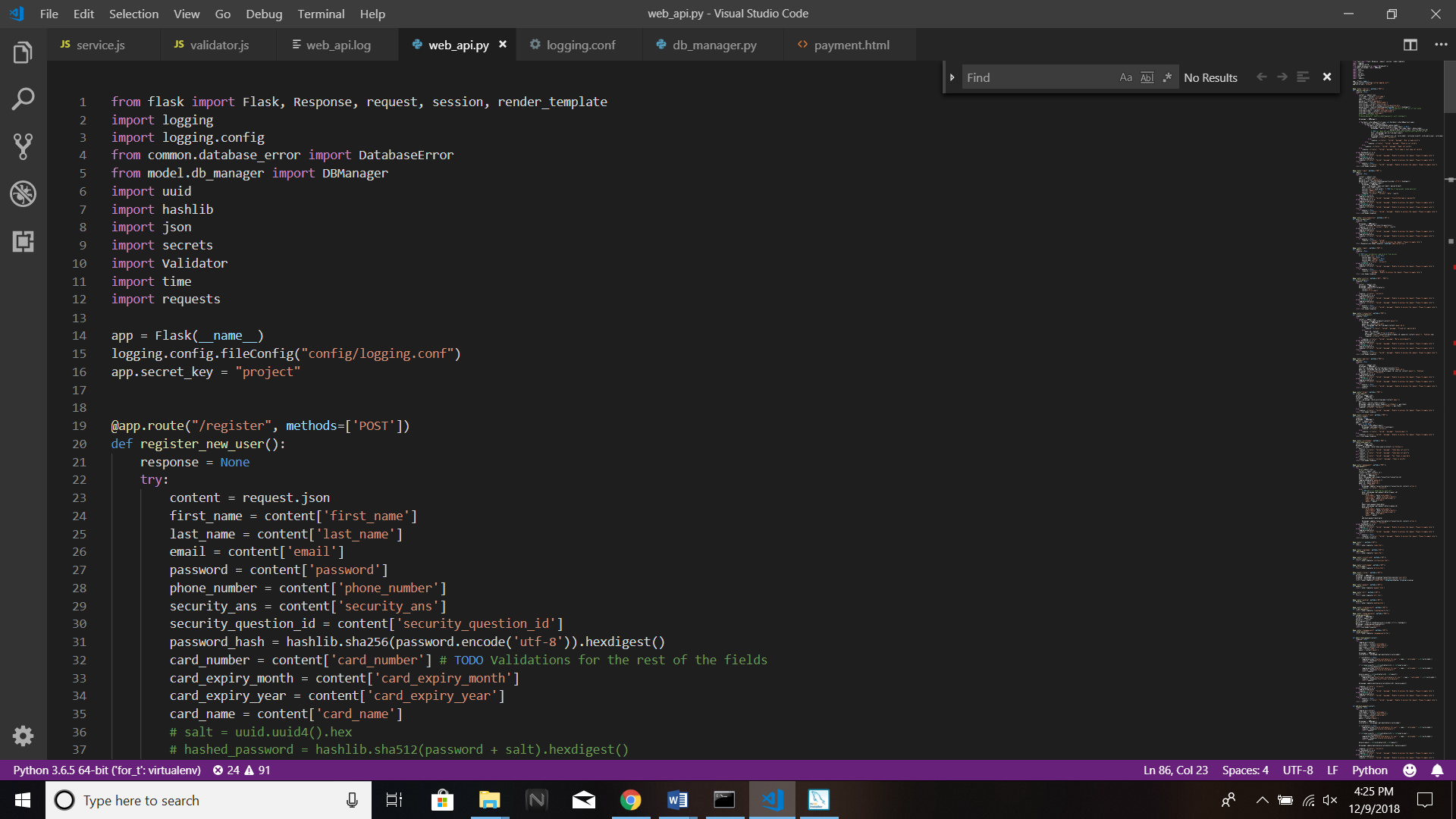
|  |
| --- |
| # has to be gmail  def isValidEmail(str):  # ends with @gmail.com  if "@gmail.com" == str[-10:]:  str = str[0:len(str)-10]  else:  return False  # 6-30 characters (not counting . or @gmail.com)  length = 0  for i in range(0, len(str)):  if (not (str[i] == '.')):  length = length + 1  if (length > 30 or length < 6):  return False  # uses only letters numbers and periods  regex = r".\*[^a-zA-Z0-9\.].\*"  if (re.match(regex, str)):  return False  # first and last char must be letter or number  if (str[0] == '.' or str[len(str)-1] == '.'):  return False  # no consecutive periods  if (re.match(r".\*\.\..\*", str)):  return False  # 8 or more chars must contain a letter  if (len(str) > 7):  regex = r".\*[a-zA-Z].\*"  if (not re.match(regex, str)):  return False  return True |

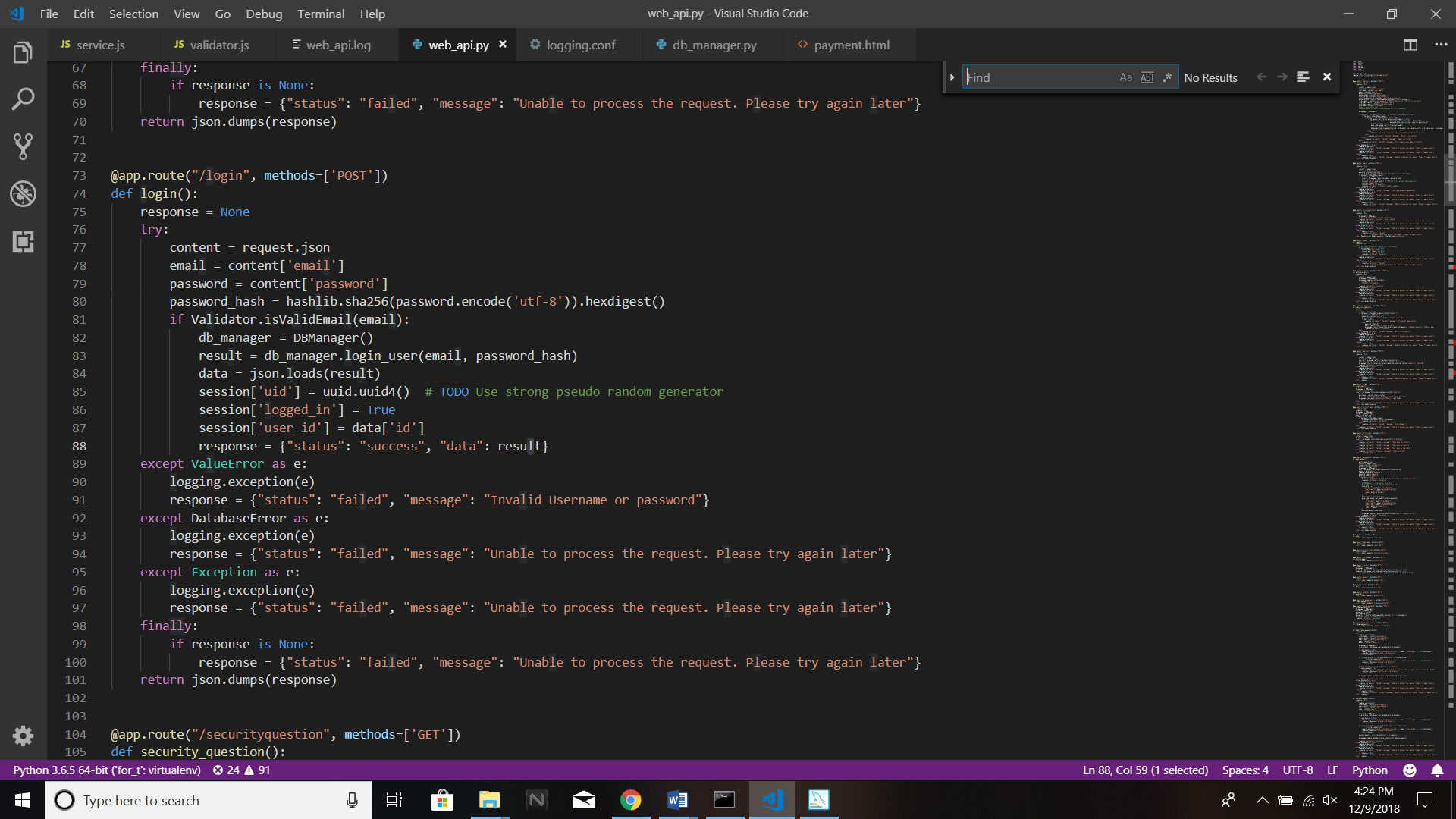
# Use of centralized logging and logging of various application events

## Sample Code

* Use of centralized logging and logging of various application events

Errors are logged to the web\_api.log file; the user is notified, but not in a descriptive way.





# List all events that the application specifically audits

* Any Request to the server.
* Entering and exiting methods to trace.
* Invalid Logon Attempts
* Any Errors along with stack trace and messages.

# Exception/error handling

|  |
| --- |
| def add\_bank\_payment(content):  try:  card\_number = content['card\_number']  expiry\_month = content['expiry\_month']  expiry\_year = content['expiry\_year']  name = content['name']  amount = content['amount']  db\_manager = DBManager()  card\_details = db\_manager.get\_bank\_details(card\_number)  if card\_details is None:  raise ValueError("Invalid card details")  if int(expiry\_month) != int(card\_details[4]) or int(expiry\_year) \  != int(card\_details[5]):  raise ValueError("Invalid card details")  balance\_amount = int(card\_details[6] + int(amount))  if balance\_amount < 0:  raise ValueError("Insufficient card balance")  db\_manager.update\_bank\_balance(card\_details[0], balance\_amount)  response = {"status": "success"}  except DatabaseError as e:  logging.exception(e)  raise ValueError("Unable to process the request")  except Exception as e:  logging.exception(e)  raise ValueError("Unable to process the request")  return True |

|  |
| --- |
| @app.route("/register", methods=['POST'])  def register\_new\_user():  response = None  try:  content = request.json  first\_name = content['first\_name']  last\_name = content['last\_name']  email = content['email']  password = content['password']  phone\_number = content['phone\_number']  security\_ans = content['security\_ans']  security\_question\_id = content['security\_question\_id']  password\_hash = hashlib.sha256(password.encode('utf-8')).hexdigest()  card\_number = cipher\_suite.encrypt(content['card\_number'].encode())  card\_expiry\_month = content['card\_expiry\_month']  card\_expiry\_year = content['card\_expiry\_year']  card\_name = content['card\_name']  # salt = uuid.uuid4().hex  # hashed\_password = hashlib.sha512(password + salt).hexdigest()  db\_manager = DBManager()  if Validator.isValidName(first\_name) and Validator.isValidName(last\_name):  if Validator.isValidEmail(email):  if Validator.isValidPhoneNumber(phone\_number):  if db\_manager.get\_profile\_details\_email(email) is None:  db\_manager.register\_user(first\_name, last\_name, email, phone\_number,  password\_hash, security\_ans, security\_question\_id)  # TODO Call Bank API and verify card details before inserting into our db.  data = db\_manager.get\_id\_from\_email(email)  user\_id = data[0]  db\_manager.insert\_payment(user\_id, card\_number, card\_expiry\_month, card\_expiry\_year, card\_name)  response = {"status": "success"}  else:  response = {"status": "failed", "message": "User already exist"}  else:  response = {"status": "failed", "message": "Phone no not valid"}  else:  response = {"status": "failed", "message": "Email not valid"}  else:  response = {"status": "failed", "message": "First name or Last name not valid"}  except DatabaseError as e:  logging.exception(e)  response = {"status": "failed", "message": "Unable to process the request. Please try again later"}  except Exception as e:  logging.exception(e)  response = {"status": "failed", "message": "Unable to process the request. Please try again later"}  finally:  if response is None:  response = {"status": "failed", "message": "Unable to process the request. Please try again later"}  return json.dumps(response) |

# Sample log file entries

Below is an invalid login attempt:

|  |
| --- |
| 2018-12-09 20:45:20,799 - root - INFO - db\_manager : 443 - Entered login\_user  2018-12-09 20:45:20,801 - root - ERROR - web\_api : 99 - Invalid Username or Password  Traceback (most recent call last):  File "web\_api.py", line 88, in login  result = db\_manager.login\_user(email, password\_hash)  File "D:\Courses\Fall 18\Secure Programming\Project\SP-GRP-4\model\db\_manager.py", line 454, in login\_user  raise ValueError("Invalid Username or Password")  ValueError: Invalid Username or Password  2018-12-09 20:45:20,801 - root - INFO - web\_api : 100 - Invalid Logon attempt forpaullewislobo@gmail.com |

Other Logs

|  |
| --- |
| 2018-12-09 20:41:13,344 - werkzeug - INFO - \_internal : 88 - 127.0.0.1 - - [09/Dec/2018 20:41:13] "POST /makepayment HTTP/1.1" 200 –  2018-12-09 20:40:50,765 - root - INFO - db\_manager : 303 - Entered get\_incoming\_transactions  2018-12-09 20:40:50,766 - root - INFO - db\_manager : 312 - Exited get\_single\_transactions  2018-12-09 20:40:50,779 - root - INFO - db\_manager : 422 - Entered get\_payment\_details  2018-12-09 20:40:50,780 - root - INFO - db\_manager : 432 - Exited get\_payment\_details  2018-12-09 20:40:50,791 - root - INFO - db\_manager : 480 - Entered get\_bank\_details  2018-12-09 20:40:50,792 - root - INFO - db\_manager : 490 - Exited get\_bank\_details  2018-12-09 20:40:50,805 - root - INFO - db\_manager : 501 - Entered update\_bank\_balance  2018-12-09 20:40:50,860 - root - INFO - db\_manager : 510 - Exited update\_bank\_details  2018-12-09 20:40:50,872 - root - INFO - db\_manager : 422 - Entered get\_payment\_details  2018-12-09 20:40:50,873 - root - INFO - db\_manager : 432 - Exited get\_payment\_details  2018-12-09 20:40:50,885 - root - INFO - db\_manager : 480 - Entered get\_bank\_details  2018-12-09 20:40:50,886 - root - INFO - db\_manager : 490 - Exited get\_bank\_details  2018-12-09 20:40:50,897 - root - INFO - db\_manager : 501 - Entered update\_bank\_balance  2018-12-09 20:40:51,268 - root - INFO - db\_manager : 510 - Exited update\_bank\_details  2018-12-09 20:40:51,305 - root - INFO - db\_manager : 327 - Entered update\_transaction\_details  2018-12-09 20:40:51,309 - root - INFO - db\_manager : 333 - Exited update\_transaction\_details |

# Session management:

Session id created using cryptographically secure function.

|  |
| --- |
| session['uid'] = secrets.token\_urlsafe(256) |

It is created on ever login attempt so it cannot be reused.

|  |
| --- |
| @app.route("/login", methods=['POST'])  def login():  response = None  try:  content = request.json  email = content['email']  password = content['password']  password\_hash = hashlib.sha256(password.encode('utf-8')).hexdigest()  if Validator.isValidEmail(email):  db\_manager = DBManager()  result = db\_manager.login\_user(email, password\_hash)  data = json.loads(result)  session['uid'] = secrets.token\_urlsafe(256)  session['logged\_in'] = True  session['user\_id'] = data['id']  if int(data['invalid\_password']) >= 3:  response = {"status": "failed", "message": "Account Locked."}  else:  db\_manager.valid\_login(email)  response = {"status": "success", "data": result}  except ValueError as e:  logging.exception(e)  logging.info("Invalid Logon attempt for" + email)  db\_manager = DBManager()  db\_manager.invalid\_login(email)  response = {"status": "failed", "message": "Invalid Username or password"}  except DatabaseError as e:  logging.info("Invalid Logon attempt for" + email)  logging.exception(e)  db\_manager = DBManager()  db\_manager.invalid\_login(email)  response = {"status": "failed", "message": "Invalid Username or password"}  except Exception as e:  logging.info("Invalid Logon attempt for" + email)  logging.exception(e)  db\_manager = DBManager()  db\_manager.invalid\_login(email)  response = {"status": "failed", "message": "Invalid Username or password"}  finally:  if response is None:  response = {"status": "failed", "message": "Unable to process the request. Please try again later"}  return json.dumps(response) |

On logout Session id is removed

|  |
| --- |
| session.pop('uid', None) |

# Authentication and authorization

Authentication is being done using 2 Factor Authentication

On registration and change password, the password is being encrypted and stored in the database.

On every login attempt the password is encrypted and compared with the encrypted password in the database. If an attacker gets hold of the encrypted password he will still not be able to access the application.

Authorization is being done on every request by checking the user is logged in before displaying any functionality to the user.

Login

|  |
| --- |
| @app.route("/login", methods=['POST'])  def login():  response = None  try:  content = request.json  email = content['email']  password = content['password']  password\_hash = hashlib.sha256(password.encode('utf-8')).hexdigest()  if Validator.isValidEmail(email):  db\_manager = DBManager()  result = db\_manager.login\_user(email, password\_hash)  data = json.loads(result)  session['uid'] = secrets.token\_urlsafe(256)  session['logged\_in'] = True  session['user\_id'] = data['id']  if int(data['invalid\_password']) >= 3:  response = {"status": "failed", "message": "Account Locked."}  else:  db\_manager.valid\_login(email)  response = {"status": "success", "data": result}  except ValueError as e:  logging.exception(e)  logging.info("Invalid Logon attempt for" + email)  db\_manager = DBManager()  db\_manager.invalid\_login(email)  response = {"status": "failed", "message": "Invalid Username or password"}  except DatabaseError as e:  logging.info("Invalid Logon attempt for" + email)  logging.exception(e)  db\_manager = DBManager()  db\_manager.invalid\_login(email)  response = {"status": "failed", "message": "Invalid Username or password"}  except Exception as e:  logging.info("Invalid Logon attempt for" + email)  logging.exception(e)  db\_manager = DBManager()  db\_manager.invalid\_login(email)  response = {"status": "failed", "message": "Invalid Username or password"}  finally:  if response is None:  response = {"status": "failed", "message": "Unable to process the request. Please try again later"}  return json.dumps(response) |

Encryption of password

|  |
| --- |
| password\_hash = hashlib.sha256(password.encode('utf-8')).hexdigest() |

# Cryptography

* We are using the https protocol using self-signed certificates. Ideally, we should use a certificate signed by a CA however for the purpose of this project it is not feasible.

|  |
| --- |
| if \_\_name\_\_ == "\_\_main\_\_":  csrf.init\_app(app)  app.run(ssl\_context='adhoc') |

* Encryption of various sensitive data is being done to prevent unauthorized access.

|  |
| --- |
| password\_hash = hashlib.sha256(password.encode('utf-8')).hexdigest() |

|  |
| --- |
| card\_number = cipher\_suite.encrypt(content['card\_number'].encode())  "card\_number": cipher\_suite.decrypt(data["card\_number"].encode()).decode(), |

# Privilege management

* Standard user permission required to run the server.
* No elevated privileges required for any functionality.
* No opening for attackers to run any commands to get elevated privileges.

# Mitigationsfor Buffer Overflow:

* Buffer overflows are mitigated by the scripting language python.
* Since it is not susceptible to buffer overflows, there is no added code to avoid these vulnerabilities.

# Mitigations for Cross Site Request Forgeries

* CSRF token generated and stored in hidden field at the client

|  |
| --- |
| <input type="hidden" id="csrf\_token" value="{{ csrf\_token() }}"/> |

* On API request, CSRF token added in the headers and passed to the server.

|  |
| --- |
| function transactionService(data){  let request = new XMLHttpRequest();  request.open("POST", "/makepayment", false);  let csrf\_token = document.getElementById("csrf\_token").value  request.setRequestHeader("X-CSRFToken", csrf\_token);  request.setRequestHeader('Content-Type', 'application/json; charset=UTF-8');  request.send(JSON.stringify(data));  let response = JSON.parse(request.response)  if (response["status"]=="success"){  return true  }else{  alert(response["message"]);  return false  } |

* Server validates the CSRF token for all APIs using flask\_wtf framework.

|  |
| --- |
| from flask\_wtf.csrf import CSRFProtect  csrf = CSRFProtect()  app = Flask(\_\_name\_\_)  if \_\_name\_\_ == "\_\_main\_\_":  csrf.init\_app(app)  app.run(ssl\_context='adhoc') |

# Mitigations for injection attacks

All injection attacks being taken care by input and output validations.

Input validations done on both UI and server side.

Output validations done when retrieving data from database.

UI JS Validations sample

|  |
| --- |
| /\*  validate sign up (page 1)  reference https://www.w3schools.com/js/js\_validation.asp  \*/  function validateSignUp () {  //first name  var fname = document.forms["sign-in"]["firstname"].value;  if (!isValidName (fname)) {  badThingHappened("Invalid input for first name");  return false;  }  //last name  var lname = document.forms["sign-in"]["lastname"].value;  if (!isValidName (lname)) {  badThingHappened("invalid input for last name");  return false;  }  //email  var email = document.forms["sign-in"]["email"].value;  if (!isValidEmail (email)) {  badThingHappened("Invalid input for gmail");  return false;  }  //card no  var card\_number = document.forms["sign-in"]["card\_number"].value;  if (!isValidCardNumber (card\_number)) {  badThingHappened("Invalid input for credit card");  return false;  }  //phone  var phone = document.forms["sign-in"]["phone"].value;  if (!isValidPhoneNumber (phone)) {  badThingHappened("Invalid input for phone number");  return false;  }  //password  var password = document.forms["sign-in"]["password"].value;  if (!isValidPassWord (password)) {  badThingHappened("Invalid input for passowrd");  return false;  }  //TODO: question no front end question check yet, will whitelist options  //answer (use name validator)  var answer = document.forms["sign-in"]["answer"].value;  if (!isValidName (fname)) {  badThingHappened("Invalid input for answer");  return false;  }  let data = {  "first\_name" : fname,  "last\_name" : lname,  "email" : email,  "phone\_number" : phone,  "password" : password,  "security\_ans" : answer,  "security\_question\_id": 1, //TODO Map security question id  "card\_number": document.forms["sign-in"]["card\_number"].value,  "card\_expiry\_month": document.forms["sign-in"]["card\_expiry\_month"].value,  "card\_expiry\_year": document.forms["sign-in"]["card\_expiry\_year"].value,  "card\_name": document.forms["sign-in"]["card\_name"].value  }  if (registerUser(data) == true){  window.location = "/loginpage";  }  return true;  } |

Server Validations sample

|  |
| --- |
| # Assumptions for a valid name (subject to change):  # Only letters, no more than three spaces,  # 0-2 non-consecutive or each hyphen, period, and single-quote  # length has 1-50 letters  def isValidName(str):  if (len(str) > 50 or len(str) < 1):  return False  regex = r".\*[^a-zA-Z0-9\-\.' ].\*"  if (re.match(regex, str)):  return False  if (re.match(r".\*\-\-.\*", str)):  return False  if (re.match(r".\*\.\..\*", str)):  return False  if (re.match(r".\*''.\*", str)):  return False  if (re.match(r".\* .\*", str)):  return False  if (re.match(r".\*\-.\*\-.\*\-.\*", str)):  return False  if (re.match(r".\*\..\*\..\*\..\*", str)):  return False  if (re.match(r".\*'.\*'.\*'.\*", str)):  return False  if (re.match(r".\* .\* .\* .\* .\*", str)):  return False  return True |

# Handling of sensitive information

* Encryption is done on sensitive fields before storing in the database.
* Passwords cannot be decrypted as a hash function used before storing it the database.

|  |
| --- |
| password\_hash = hashlib.sha256(password.encode('utf-8')).hexdigest() |

* Other sensitive fields like credit card number can be decrypted when being fetched from the database.

|  |
| --- |
| card\_number = cipher\_suite.encrypt(content['card\_number'].encode())  "card\_number": cipher\_suite.decrypt(data["card\_number"].encode()).decode(), |

* Logging of all sensitive data that was used for debugging was removed.
* Error messages and stack traces logged but never sent to the user.

# And any other types of mitigations/techniques that were used.

* 2 Factor Authentication.
* Disabling debugging.
* Enabled production configuration for server.