```python
# from endpoints.py
""" endpoints written by Lia Ferguson:
        /login_bypass
        /login_query
        /trojan_horse
        /suspect
        /login

        with the exception of the two try - except blocks and json.dumps()
          lines written by Andrew Fecher
        in /login_query
"""
NUM_RECORDS_USERS_TABLE = 8
bp = Blueprint('endpoints', __name__, url_prefix='/endpoints')

# endpoint for Step 1 SQL Injection Task
@bp.route('/login_bypass', methods = ['POST'])
def login_bypass():
    database = get_db()

    user_id = request.get_json()['user_id']
    password = request.get_json()['password']
    # wrong way to compose SQL query based on secure coding practices
    # this allows for SQL Injection to occur
    quote = ""
    if user_id.find("\"") == -1 and user_id.find('\'') == -1:
        user_id = "\"" + user_id + "\""
        password = "\"" + password + "\""
    elif user_id.find('\'') != -1:
        quote = "\'"
    else:
        quote = "\""

    login_q = 'SELECT * FROM USERS WHERE User_ID = {quote}{u_id} AND Password =
       {pwd}'.format(
                            quote=quote, u_id = user_id, pwd = password)

    query_result = database.execute(login_q).fetchall()
    response = {}
    if len(query_result) == NUM_RECORDS_USERS_TABLE:
        response = {
            'isQuerySuccessful': 'true',
            'status': 'SUCCESS',
            'message': 'Congratulations! You successfully used SQL Injection to
              bypass authentication.'
        }

    else:
        response = {
```

```
47              'isQuerySuccessful': 'false',
48              'status': 'ERROR',
49              'message': 'SQL Injection was not successful, please try again.'
50          }
51      print(response)
52      print(user_id)
53      return jsonify(response)
54
55  # endpoint for all SQL Injection after step 1
56  @bp.route('/login_query', methods = ['POST'])
57  def login_query():
58      database = get_db()
59
60      user_id = request.get_json()['user_id']
61      password = request.get_json()['password']
62      game_step = request.get_json()['game_step']
63
64      quote = ""
65      if user_id.find("\"") == -1 and user_id.find('\'') == -1:
66          user_id = "\"" + user_id + "\""
67          password = "\"" + password + "\""
68      elif user_id.find('\'') != -1:
69          quote = "\'"
70      else:
71          quote = "\""
72
73      login_q = 'SELECT * FROM USERS WHERE User_ID = {quote}{u_id} AND Password =
            {quote}{pwd}'.format(
74                              quote=quote, u_id = user_id, pwd = password)
75
76      commands = login_q.split(";", -1)
77      all_query_results = []
78      formatted_query_results = []
79      error = ''
80      try:
81          for command in commands:
82              query_results = database.execute(command).fetchall()
83              all_query_results.append(query_results)
84      except Exception as e:
85          error = e.args
86
87      if error == '':
88          formatted_query_results =''
89          try:
90              table_columns = queried_table_columns(commands[1])
91              formatted_query_results = format_query_results(all_query_results
                    [1], table_columns, game_step)
92          except Exception as e:
93              print(e)
```

```
94              formatted_query_results = 'ERROR'
95          match_expected_results = check_expected_results(all_query_results[1], game_step)
96      if len(formatted_query_results) > 0:
97          if match_expected_results:
98              print_results_to_file(formatted_query_results, game_step)
99              response = {
100                 'isQuerySuccessful': 'true',
101                 'correctResults': 'true',
102                 'results': json.dumps(formatted_query_results),
103                 'error': ''
104             }
105         else:
106             if len(table_columns) > len(CORRECT_RESULTS[game_step][0]):
107                 error = 'SQL Query returns too much information. Follow the directions and be more specific!'
108             else:
109                 error = 'SQL Query was valid but it doesn\'t return the information that you need!'
110             response = {
111                 'isQuerySuccessful': 'true',
112                 'correctResults': 'false',
113                 'results': json.dumps(formatted_query_results),
114                 'error': error
115             }
116     else:
117         error = error if error != '' else 'SQL Query was valid but there were no matching records returned.'
118         response = {
119             'isQuerySuccessful': 'false',
120             'correctResults': 'false',
121             'results': '',
122             'error': error
123         }
124     print(json.dumps(formatted_query_results))
125     print(response)
126     return jsonify(response)
127
128 # endpoint to trigger trojan horse process in step 5
129 @bp.route('/trojan_horse', methods = ['POST'])
130 def trojan_horse():
131     first_name = request.get_json()['first_name']
132     last_name = request.get_json()['last_name']
133
134     if first_name == '':
135         response = {
136             'isSuccess': 'false',
137             'message': 'first name must be provided in order to proceed'
138         }
```

```
139     elif last_name == '':
140         response = {
141             'isSuccess': 'false',
142             'message': 'last name must be provided in order to proceed'
143         }
144     else:
145         execute_trojan_horse(first_name, last_name)
146 
147         response = {
148             'isSuccess': 'true',
149             'message': 'Just a moment! Loading...'
150         }
151 
152     jsonify(response)
153     return response
154 
155 # endpoint that processes submission of suspect guesses
156 @bp.route('/suspect', methods = ['POST'])
157 def suspect():
158     name = request.get_json()['name']
159     game_step = request.get_json()['game_step']
160 
161     correct = check_suspect(name, game_step)
162 
163     response = {}
164     if correct:
165         print_results_to_file(name, game_step)
166         response = {
167             'correct': 'true',
168             'message': 'The evidence suggests that this person is a suspect.'
169         }
170     else:
171         response = {
172             'correct': 'false',
173             'message': 'There isn\'t enough evidence for this person to be a
                suspect.'
174         }
175 
176     jsonify(response)
177     return response
178 
179 # endpoint that processes normal login in final step of the game
180 @bp.route('/login', methods = ['POST'])
181 def login():
182     database = get_db()
183 
184     user_id = request.get_json()['user_id']
185     password = request.get_json()['password']
186     response = {}
```

```
187     if user_id == '':
188         response = {
189             'isLoginSuccessful': 'false',
190             'error': 'You must provide a username'
191         }
192     elif password == '':
193         response = {
194             'isLoginSuccessful': 'false',
195             'error': 'You must provide a password'
196         }
197 
198     quote = ""
199     formatted_password = ''
200     if user_id.find("\"") == -1 and user_id.find('\'') == -1:
201         user_id = "\"" + user_id + "\""
202         formatted_password = "\"" + password + "\""
203     elif user_id.find('\'') != -1:
204         quote = "\'"
205     else:
206         quote = "\""
207 
208     login_q = 'SELECT * FROM USERS WHERE User_ID = {quote}{u_id}' .format
          (quote=quote, u_id = user_id)
209     query_result = database.execute(login_q).fetchone()
210     record = tuple(y for y in query_result)
211     print(record)
212     if len(query_result) == 0:
213         response = {
214             'isLoginSuccessful': 'false',
215             'error': 'Invalid username provided'
216         }
217     else:
218         if password == record[1]:
219             response = {
220                 'isLoginSuccessful': 'true',
221                 'error': ''
222             }
223         else:
224             response =  {
225                 'isLoginSuccessful': 'false',
226                 'error': 'Invalid password provided'
227             }
228     print(response)
229     return response
230 
231 # from database_functions.py
232 # lines 6-33 written by Lia Ferguson
233 # lines 49-91 written by Tom Chmura
234 # dictionary that maps table name to the path of the csv data to populate it
```

```
DB_TABLE_DICT = {
    'BUILDING_ACCESS': 'app/data/BUILDING_ACCESS.csv',
    'COMPUTER_ACCESS': 'app/data/COMPUTER_ACCESS.csv',
    'COMPUTER_TERMINALS': 'app/data/COMPUTER_TERMINALS.csv',
    'QUESTIONNAIRE': 'app/data/QUESTIONNAIRE.csv',
    'USER_INFO': 'app/data/USER_INFO.csv',
    'USERS': 'app/data/USERS.csv',
    'PURCHASE_ORDERS': 'app/data/PURCHASE_ORDERS.csv'
}

# read in data for csv files, and format records properly for insertion into DB
# proper format needed = list of tuples with data in order by columns
# ex. for USERS table
#     return = [(1234, password), (2345, pwd123)]
def get_initial_data(database):
    # path of USERS table csv data
    users_data_path = DB_TABLE_DICT['USERS']
    # SQL query to insert records into users table
    users_insert_query = 'INSERT into USERS (User_ID, Password) VALUES (? , ?)'
    # read in csv data
    with open(users_data_path, newline='\n') as csvfile:
        user_data = csv.reader(csvfile, delimiter=',')
        # skip header row
        next(user_data, None)
        # insert records into database
        for record in user_data:
            database.execute(users_insert_query, record)
            database.commit()

    # path of QUESTIONNAIRE table csv data
    questionnaire_data_path = DB_TABLE_DICT['QUESTIONNAIRE']
    # SQL query to insert records into questionnaire table
    questionnaire_insert_query = 'INSERT into QUESTIONNAIRE (User_ID,
      Favorite_food, Favorite_hobby, Favorite_drink, Allergies) VALUES
      (? , ? , ? , ?, ?)'
    # read in csv data
    with open(questionnaire_data_path, newline='\n') as csvfile:
        questionnaire_data = csv.reader(csvfile, delimiter=',')
        # skip header row
        next(questionnaire_data, None)
        # insert records into database
        for record in questionnaire_data:
            database.execute(questionnaire_insert_query, record)
            database.commit()

    # path of USER_INFO table csv data
    userinfo_data_path = DB_TABLE_DICT['USER_INFO']
    # SQL query to insert records into user info table
    userinfo_insert_query = 'INSERT into USER_INFO (User_ID, First_name,
```

```
          Last_name, Superhero_Name) VALUES (? , ? , ? , ?)'
    # read in csv data
    with open(userinfo_data_path, newline='\n') as csvfile:
        userinfo_data = csv.reader(csvfile, delimiter=',')
        # skip header row
        next(userinfo_data, None)
        # insert records into database
        for record in userinfo_data:
            database.execute(userinfo_insert_query, record)
            database.commit()


    # path of PURCHASE_ORDERS table csv data
    purchaseorders_data_path = DB_TABLE_DICT['PURCHASE_ORDERS']
    # SQL query to insert records into purchase orders table
    purchaseorders_insert_query = 'INSERT into PURCHASE_ORDERS (Po_number,
      User_ID, Item, Cost, Time_received) VALUES (? , ? , ? , ? , ?)'
    # read in csv data
    with open(purchaseorders_data_path, newline='\n') as csvfile:
        purchaseorders_data = csv.reader(csvfile, delimiter=',')
        # skip header row
        next(purchaseorders_data, None)
        # insert records into database
        for record in purchaseorders_data:
            database.execute(purchaseorders_insert_query, record)
            database.commit()


    # path of BUILDING_ACCESS table csv data
    buildingaccess_data_path = DB_TABLE_DICT['BUILDING_ACCESS']
    # SQL query to insert records into building access table
    buildingaccess_insert_query = 'INSERT into BUILDING_ACCESS (Building_ID,
      Building_time, User_ID) VALUES (? , ? , ? )'
    # read in csv data
    with open(buildingaccess_data_path, newline='\n') as csvfile:
        buildingaccess_data = csv.reader(csvfile, delimiter=',')
        # skip header row
        next(buildingaccess_data, None)
        # insert records into database
        for record in buildingaccess_data:
            database.execute(buildingaccess_insert_query, record)
            database.commit()

# from schema.sql
#Written by Tom Chmura
#small updates to USER_INFO, QUESTIONNAIRE, BUILDING_ACCESS, and
  PURCHASE_ORDERS tables by Lia Ferguson

-- Table: BUILDING_ACCESS
```

```
327 CREATE TABLE BUILDING_ACCESS(
328 Building_ID INT NOT NULL,
329 Building_time TIME NOT NULL,
330 User_ID INT NOT NULL,
331 FOREIGN KEY(User_ID) references USERS(User_ID));
332 
333 -- Table: QUESTIONNAIRE
334 CREATE TABLE QUESTIONNAIRE(
335 User_ID INT NOT NULL,
336 Favorite_food VARCHAR(30),
337 Favorite_drink VARCHAR(30),
338 Favorite_hobby VARCHAR(30),
339 Allergies VARCHAR(30),
340 PRIMARY KEY(User_ID)
341 FOREIGN KEY(User_ID) references USERS(User_ID));
342 
343 -- Table: USER_INFO
344 CREATE TABLE USER_INFO(
345 User_ID INT NOT NULL,
346 First_name VARCHAR(20) NOT NULL,
347 Last_name VARCHAR(20),
348 Superhero_Name VARCHAR(30),
349 PRIMARY KEY(User_ID)
350 FOREIGN KEY(User_ID) references USERS(User_ID));
351 
352 -- Table: USERS
353 CREATE TABLE USERS(
354 User_ID INT NOT NULL,
355 Password VARCHAR(30) NOT NULL,
356 PRIMARY KEY(User_ID));
357 
358 -- Table: PURCHASE_ORDERS
359 CREATE TABLE PURCHASE_ORDERS (
360 PO_NUMBER     INT          NOT NULL,
361 USER_ID       INT          NOT NULL,
362 ITEM          VARCHAR (30) NOT NULL,
363 COST          DOUBLE       NOT NULL,
364 TIME_RECEIVED TIME,
365 PRIMARY KEY (PO_NUMBER)
366 FOREIGN KEY(User_ID) references USERS(User_ID));
367 
368 # from helper_functions.py
369 """ code written by Lia Ferguson:
370         all code besides the code written by Andrew Fecher
371 """
372 """code written by Andrew Fecher:
373         line 18, lines 106-117
374 """
375 
```

```
376 # Data structures to hold columns of data tab;es
377 USERS_COLUMNS = ['User_ID', 'Password']
378 USER_INFO_COLUMNS = ['User_ID', 'First_name', 'Last_name', 'Superhero_Name']
379 QUESTIONNAIRE_COLUMNS = ['User_ID', 'Favorite_food', 'Favorite_hobby',
      'Favorite_drink', 'Allergies']
380 PURCHASE_ORDERS_COLUMNS = ['Po_number', 'User_ID', 'Item', 'Cost',
      'Time_Received']
381 BUILDING_ACCESS_COLUMNS = ['Building_ID', 'Building_time', 'User_ID']
382 SQLITE_MASTER_COLUMNS = ['type', 'name', 'tabl_name', 'rootpage', 'sql']
383 #Data structure to hold expected SQL Results
384 CORRECT_RESULTS = {
385     'S3_B1': [('BUILDING_ACCESS',), ('QUESTIONNAIRE',), ('USER_INFO',),
        ('USERS',), ('PURCHASE_ORDERS',)],
386     'S4_B1': [(12592, 'Tony', 'Stark'),
387                         (15687, 'Natasha', 'Romanoff'),
388                         (15685, 'Scott', 'Lang'),
389                         (15972, 'Peter', 'Parker'),
390                         (15423, 'Steve', 'Rogers'),
391                         (15976, 'Thanos', ''),
392                         (17896, 'Bruce', 'Banner')],
393     'S4_B2': [('steak', 'stand-up comedy', 'coffee', 'almonds')],
394     'S5_B1': [(15687, 'almonds'), (17896, 'almonds')],
395     'S5_S': ['Natasha Romanoff', 'Bruce Banner'],
396     'S6_B1': [('Building_ID',), ('Building_time',), ('User_ID',)],
397     'S6_B2': [(15687, '12:55 pm'), (17896, '12:40 pm')],
398     'S6_B3': [(15972, '10:30 am'), (15976, '11:00 am')],
399     'S6_S': ['Peter Parker', 'Thanos'],
400     'S7_B1': [(156834, 15972, 'Coffee Creamer', 5.12, '5:00 pm'),
401                         (156853, 15976, 'Almond Coffee Creamer', 5.23, '5:00
                        pm'),
402                         (438657, 15972, 'Popcorn', 10.25,'12:00pm')],
403     'S7_S': ['Thanos'],
404     'S7_B2': [(15976, 'IAmInevitable')]
405 }
406
407 # Parses out columns that are involved in the SQL Injection Query passed in by
      player
408 def queried_table_columns(query):
409     columns = []
410     columns_and_indices = {} # list that keeps track of column ordering in the
        query
411     if query.casefold().find('questionnaire') != -1:
412         if query.find('*') != -1:
413             columns = QUESTIONNAIRE_COLUMNS
414         else:
415             for column in QUESTIONNAIRE_COLUMNS:
416                 query = query.casefold().partition('from')[0]
417                 if query.find(column.casefold()) != -1:
418                     columns_and_indices[query.find(column.casefold())] = column
```

```
            indices = list(columns_and_indices.keys())
            indices.sort()
            for index in indices:
                columns.append(columns_and_indices[index])
    elif query.casefold().find('user_info') != -1:
        if query.find('*') != -1:
            columns = USER_INFO_COLUMNS
        else:
            for column in USER_INFO_COLUMNS:
                query = query.casefold().partition('from')[0]
                if query.find(column.casefold()) != -1:
                    columns_and_indices[query.find(column.casefold())] = column
            indices = list(columns_and_indices.keys())
            indices.sort()
            for index in indices:
                columns.append(columns_and_indices[index])
    elif query.casefold().find('users') != -1:
        if query.find('*') != -1:
            columns = USERS_COLUMNS
        else:
            for column in USERS_COLUMNS:
                query = query.casefold().partition('from')[0]
                if query.find(column.casefold()) != -1:
                    columns_and_indices[query.find(column.casefold())] = column
            indices = list(columns_and_indices.keys())
            indices.sort()
            for index in indices:
                columns.append(columns_and_indices[index])
    elif query.casefold().find('purchase_orders') != -1:
        if query.find('*') != -1:
            columns = PURCHASE_ORDERS_COLUMNS
        else:
            for column in PURCHASE_ORDERS_COLUMNS:
                query = query.casefold().partition('from')[0]
                if query.find(column.casefold()) != -1:
                    columns_and_indices[query.find(column.casefold())] = column
            indices = list(columns_and_indices.keys())
            indices.sort()
            for index in indices:
                columns.append(columns_and_indices[index])
    elif query.casefold().find('building_access') != -1:
        if query.find('*') != -1:
            columns = BUILDING_ACCESS_COLUMNS
        else:
            for column in BUILDING_ACCESS_COLUMNS:
                query = query.casefold().partition('from')[0]
                if query.find(column.casefold()) != -1:
                    columns_and_indices[query.find(column.casefold())] = column
            indices = list(columns_and_indices.keys())
```

```
468             indices.sort()
469             for index in indices:
470                 columns.append(columns_and_indices[index])
471     elif query.casefold().find('sqlite_master') != -1:
472         if query.find('*') != -1:
473             columns = SQLITE_MASTER_COLUMNS
474         else:
475             for column in SQLITE_MASTER_COLUMNS:
476                 query = query.casefold().partition('from')[0]
477                 if query.find(column.casefold()) != -1:
478                     columns_and_indices[query.find(column.casefold())] = column
479             indices = list(columns_and_indices.keys())
480             indices.sort()
481             for index in indices:
482                 columns.append(columns_and_indices[index])
483 
484     return columns
485 
486 # Formats query results nicely from sqlite3 data structures into dictionaries
487 # for later JSON parsing
488 def format_query_results(query_results, table_columns, game_step):
489     formatted_results = []
490     records = [tuple(y for y in row) for row in query_results]
491     print(records)
492     if len(table_columns) == 0:
493         for record in records:
494             format_record = {}
495             if game_step == 'S6_B1':
496                 format_record['Column'] = record[0]
497             formatted_results.append(format_record)
498     else:
499         for record in records:
500             format_record = {}
501             i = 0
502             for item in record:
503                 format_record[table_columns[i]] = item
504                 i += 1
505             formatted_results.append(format_record)
506     return formatted_results
507 
508 # check whether or not returned records from
509 # SQL query match the expected output
510 def check_expected_results(query_results, game_step):
511     matches_correct_results = False
512     correct_results = CORRECT_RESULTS[game_step]
513     correct_results_length = len(correct_results)
514     comparison = [] # index corresponds to record number, 1 = same, 0 =
        different
515     records = [tuple(y for y in row) for row in query_results]
```

```
516     if len(records) == correct_results_length:
517         for i in range(0, len(records)):
518             sum_match = 0
519             for j in range(0, len(records[0])):
520                 if correct_results[i][j] in records[i]:
521                     sum_match += 1
522             if sum_match == len(records[i]):
523                 comparison.append(1)
524             else:
525                 comparison.append(0)
526         print(comparison)
527         sum_comparison = 0
528         for num in comparison:
529             sum_comparison += num
530         print(sum_comparison)
531         if sum_comparison == len(correct_results):
532             matches_correct_results = True
533     return matches_correct_results
534 
535 # Print results of SQL Injection to Clues.txt file to
536 # assist player with game play
537 def print_results_to_file(formatted_results, game_step):
538     path = os.path.expanduser("~")
539     rest_of_path = ''
540         if platform.system() == 'Windows' :
541         rest_of_path = '\Desktop\SQL-Mystery-Game-Files'
542         clues='\Clues.txt'
543     else:
544         rest_of_path = '/Desktop/Sql-Mystery-Game-Files/'
545         clues='Clues.txt'
546     path += rest_of_path
547     if not os.path.isdir(path):
548         os.mkdir(path)
549     f = open(path + clues, 'a')
550     if game_step == 'S4_B1':
551         f.write("STEP 4 CLUES\n")
552         f.write("Employee User ID\'s\n\n")
553     elif game_step == 'S4_B2':
554         f.write("Tony Stark's Questionnaire Data\n\n")
555     elif game_step == 'S5_B1':
556         f.write("STEP 5 CLUES\n")
557         f.write("Discover Possible Almond Snackers\n\n")
558     elif game_step == 'S5_S':
559         f.write("Suspect\n\n")
560     f.write(json.dumps(formatted_results, indent=4, sort_keys=False))
561     f.write('\n\n')
562     f.close()
563 
564 # execute "trojan horse" - download
```

```
565 # confession file onto player's computer with their name filled in
566 def execute_trojan_horse(first_name, last_name):
567     path = os.path.expanduser("~")
568     rest_of_path = ''
569     if(platform.system() == 'Windows'):
570         rest_of_path = '\Desktop\SQL-Mystery-Game-Files\Confidential'
571         file_path = '\For Police.txt'
572     else:
573         rest_of_path = '/Desktop/Sql-Mystery-Game-Files/Confidential'
574         file_path = '/For Police.txt'
575     path += rest_of_path
576 
577     os.mkdir(path)
578     f_read = open('app/data/trojan_horse_confess_template.txt', 'r')
579     f_write = open(path + file_path, 'a')
580     f_write.write(f_read.read())
581     f_write.write(first_name + " " + last_name)
582 
583     f_read.close()
584     f_write.close()
585 
586 # verify if the suspect entered by player is correct
587 def check_suspect(name, game_step):
588     correct = False
589     suspects = CORRECT_RESULTS[game_step]
590     for suspect in suspects:
591         if name.casefold() == suspect.casefold():
592             correct = True
593             break
594     return correct
595 
```