Secure Software Development

Team 4 **ISS Testing Output**

25 October 2021

Team Members

Name	Email	ePortfolio
Andrey Smirnov	soundmaven@gmail.com	https://soundmaven.github.io/e-portfolio/
Michael Justus	micjustus@gmail.com	https://micjustus.github.io/essex-eport2/
Taylor Edgell	tayloredgell@googlemail.com	https://tedgell.github.io/MSc-ComputerScience/

Document Version History

Version	Date	Author	Details
0.1	22/10/21	Taylor Edgell	Document creation

Definitions and Abbreviations

Acronym	Description
API	Application Programming Interface
ISS	International Space Station
MVC	Model View Controller
RBAC	Role-Based Access Control
REST	Representational State Transfer
MVP	Model View Presenter
MVVM	Model View ViewModel
UML	Unified Modelling Language

Contents

1. Overview

The solution we have designed is a Safety Entry Database for the International Space Station (ISS). Throughout code development we have tested each component to ensure that it is working correctly and as intended. This has included ensuring that all security processes and function have been tested to work as intended.

The testing philosophy has been to follow both a Whitebox and Blackbox approach. Blackbox testing has taken place by providing expected and unexpected inputs to the UI, whereas Whitebox testing has been tested by running individual code components to ensure they are functioning correctly (Umar, 2019).

One key factor has been to ensure the testability of our code throughout the development cycle. This has allowed us to easily confirm that is working as required.

Additional applications such as flake8 and McCabe have been used to ensure common coding standards have been met, such as PEP 8 (Van Rossum et al, 2001), and that cyclomatic complexity has also been confirmed.

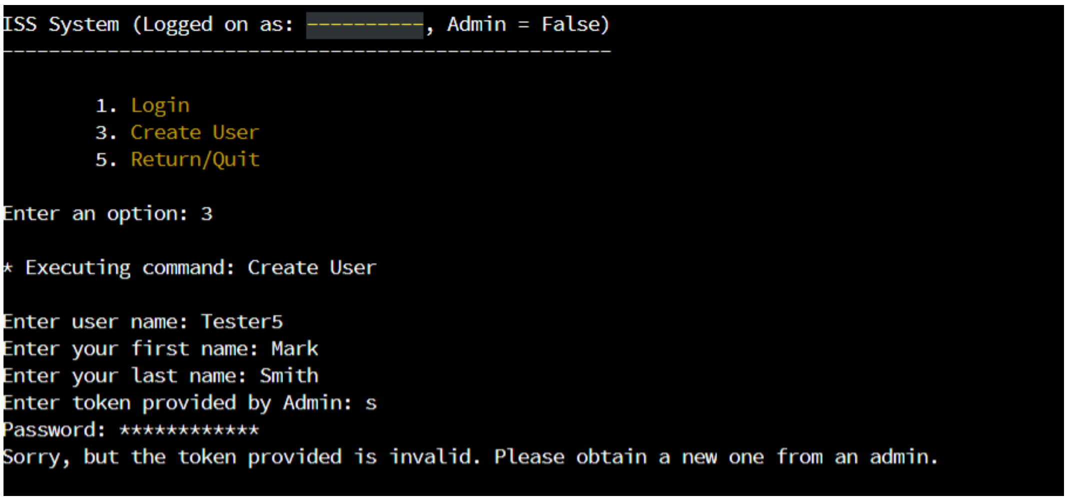
Although this document will not document every test that has taken place, a subset of tests will be documented to show the methodology that we have followed. The tests primary demonstrated within this document will be focused on the security aspects of the solution rather than the actual operations. Many of the security aspects have be designed to specifically combat common weaknesses seen in the OWASP top ten (Mitre, 2017).

2. Monolithic

2.1 Login Token

The login token allows a user to initially crate an account. An admin initially needs to generate a token, and provide it to a user, in order for them to successfully create an account.

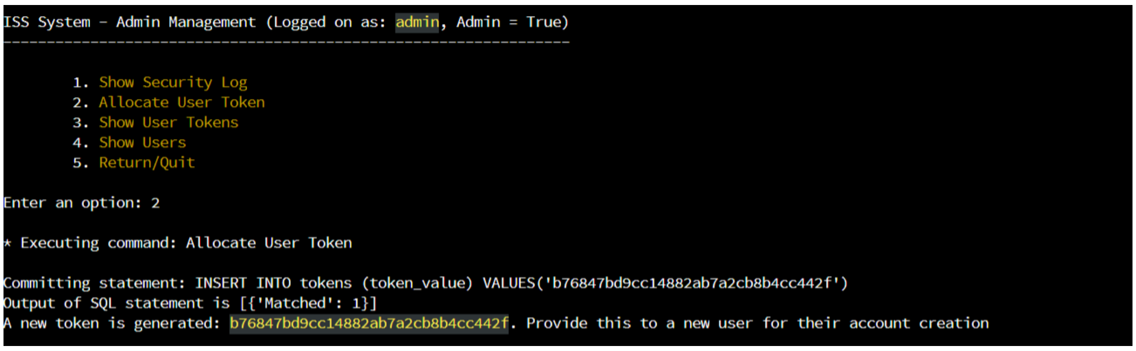
Initially we can test that this feature does now work erroneously by providing an incorrect token into the create use field. This can be seen from Figure 1.



```
ISS System (Logged on as: ----------, Admin = False)
----------------------------------------------------

        1. Login
        3. Create User
        5. Return/Quit

Enter an option: 3

* Executing command: Create User

Enter user name: Tester5
Enter your first name: Mark
Enter your last name: Smith
Enter token provided by Admin: s
Password: ************
Sorry, but the token provided is invalid. Please obtain a new one from an admin.
```

Figure 1 - Invalid Access Token

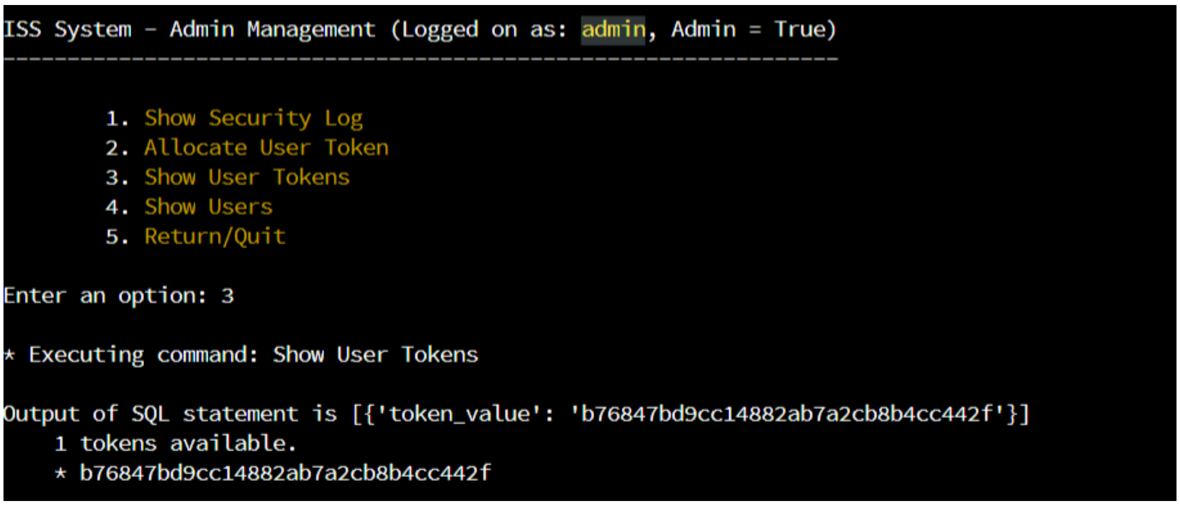
To confirm that token generation is working we must initially log in as an “Admin” class user and navigate to the administrative controls screen. It should be noted that as another security aspect the administrative controls screen is only available and visible if accessed by an “admin” class user. One in the administrative control, an access token can be generated as seen from Figure 2.



```
ISS System - Admin Management (Logged on as: admin, Admin = True)
-----------------------------------------------------------------

        1. Show Security Log
        2. Allocate User Token
        3. Show User Tokens
        4. Show Users
        5. Return/Quit

Enter an option: 2

* Executing command: Allocate User Token

Committing statement: INSERT INTO tokens (token_value) VALUES('b76847bd9cc14882ab7a2cb8b4cc442f')
Output of SQL statement is [{'Matched': 1}]
A new token is generated: b76847bd9cc14882ab7a2cb8b4cc442f. Provide this to a new user for their account creation
```

Figure 2 - Token Generation

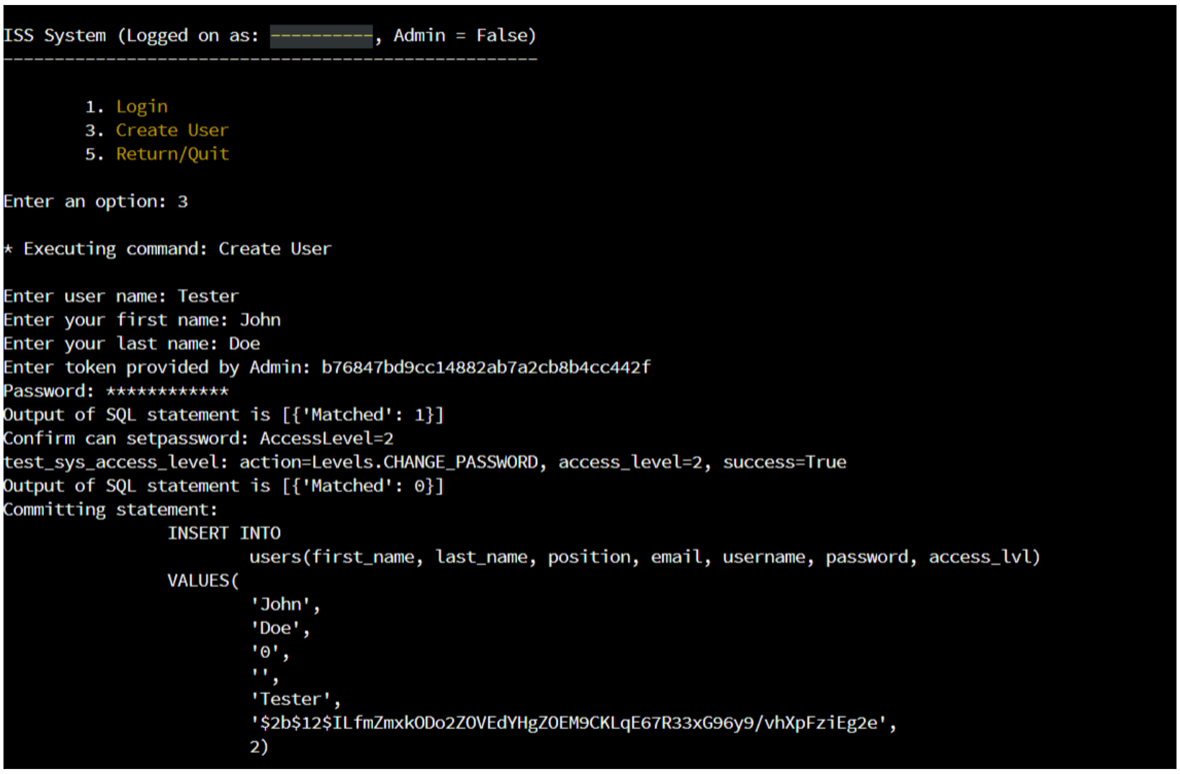
The token is then added to the list of allocated tokens as seen from Figure 3



```
ISS System - Admin Management (Logged on as: admin, Admin = True)
-----------------------------------------------------------------


        1. Show Security Log
        2. Allocate User Token
        3. Show User Tokens
        4. Show Users
        5. Return/Quit

Enter an option: 3

* Executing command: Show User Tokens

Output of SQL statement is [{'token_value': 'b76847bd9cc14882ab7a2cb8b4cc442f'}]
    1 tokens available.
    * b76847bd9cc14882ab7a2cb8b4cc442f
```

Figure 3 - Token List

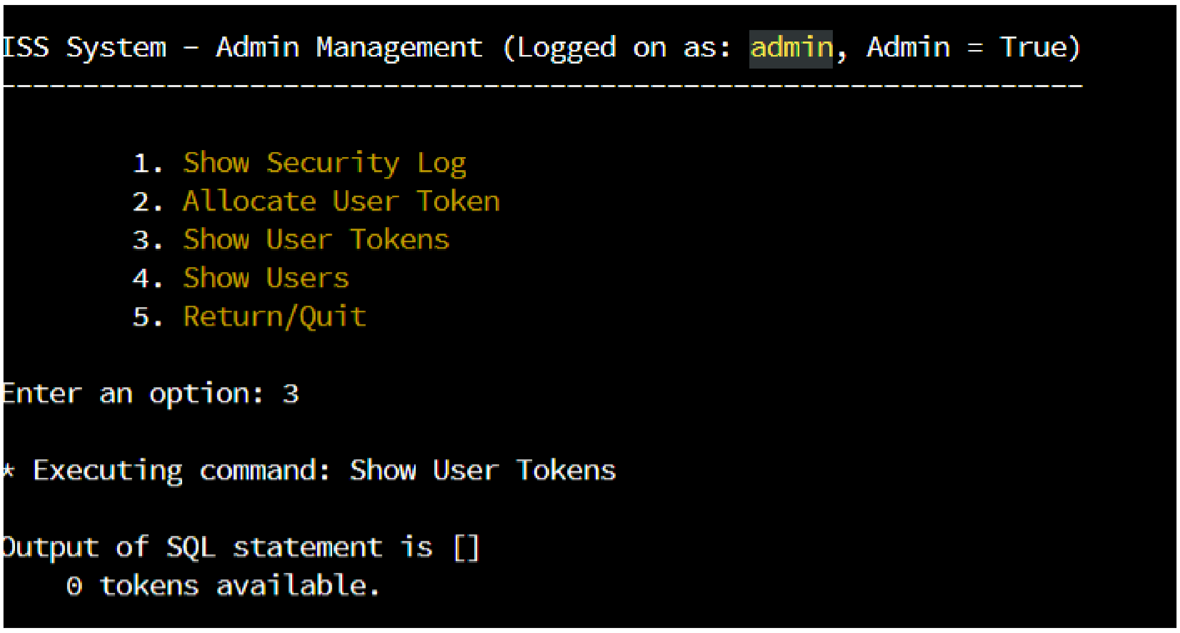
A person wanting to create an account can now use this token to proceed with account creation. This can be seen from Figure 4.



```
ISS System (Logged on as: ----------, Admin = False)
----------------------------------------------------


        1. Login
        3. Create User
        5. Return/Quit

Enter an option: 3

* Executing command: Create User

Enter user name: Tester
Enter your first name: John
Enter your last name: Doe
Enter token provided by Admin: b76847bd9cc14882ab7a2cb8b4cc442f
Password: ************
Output of SQL statement is [{'Matched': 1}]
Confirm can setpassword: AccessLevel=2
test_sys_access_level: action=Levels.CHANGE_PASSWORD, access_level=2, success=True
Output of SQL statement is [{'Matched': 0}]
Committing statement:
                INSERT INTO
                        users(first_name, last_name, position, email, username, password, access_lvl)
                VALUES(
                        'John',
                        'Doe',
                        '0',
                        '',
                        'Tester',
                        '$2b$12$ILfmZmxkODo2ZOVEdYHgZOEM9CKLqE67R33xG96y9/vhXpFziEg2e',
                        2)
```

Figure 4 - Account Creation

Once a token has been used to create an account it is then deleted from the system. This can be tested to work correctly by again logging in as an admin and accessing administrative control to look at the list of active tokens. This can be seen for Figure 5.



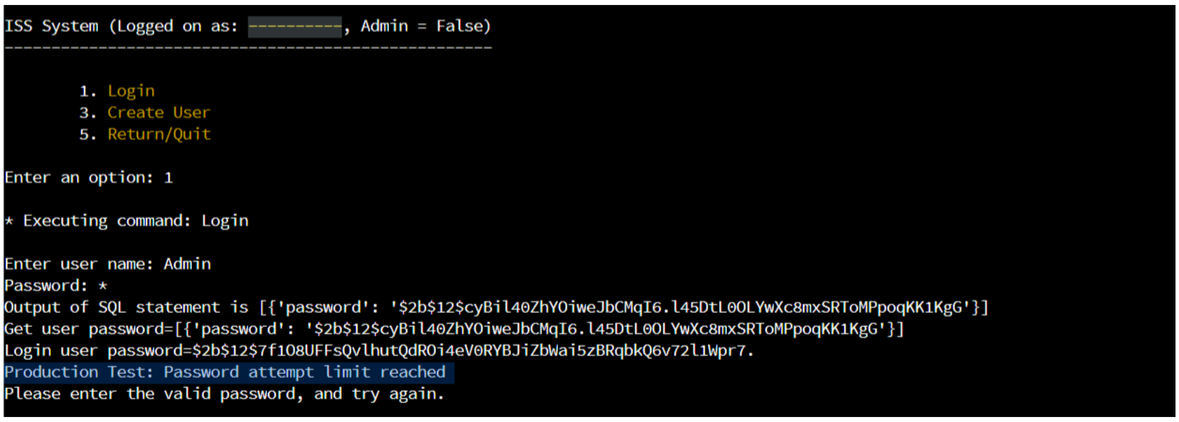
```
ISS System - Admin Management (Logged on as: admin, Admin = True)
-----------------------------------------------------------------

        1. Show Security Log
        2. Allocate User Token
        3. Show User Tokens
        4. Show Users
        5. Return/Quit

Enter an option: 3

* Executing command: Show User Tokens

Output of SQL statement is []
    0 tokens available.
```

Figure 5 - Token Removed from List

2.2 *Incorrect password limit*

Within the solution an incorrect password limit of three attempts has been set. The system will also log all incorrect attempts against a user profile while the system is running. In a real system we would set the solution to lock the user account for a day after three incorrect login attempts, but as this is a development project, we are simply allowing the system to provide a message that three incorrect logins have been made.

The testing of incorrect login attempts can be seen from Figure 6.

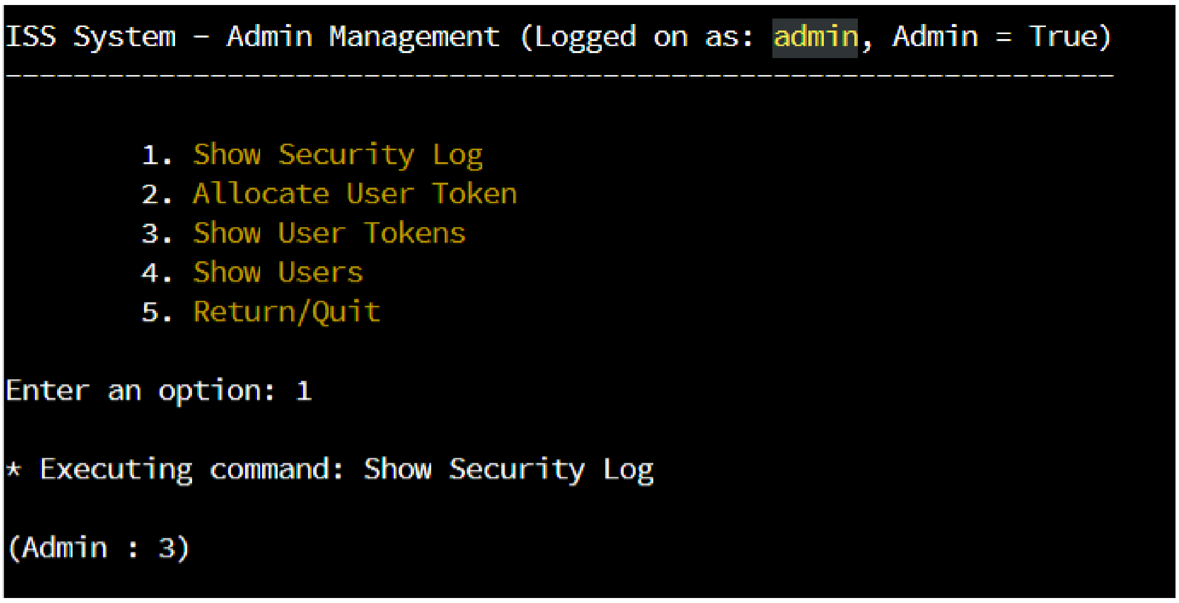


```
ISS System (Logged on as: ----------, Admin = False)
----------------------------------------------------


        1. Login
        3. Create User
        5. Return/Quit

Enter an option: 1

* Executing command: Login

Enter user name: Admin
Password: *
Output of SQL statement is [{'password': '$2b$12$cyBil40ZhYOiweJbCMqI6.l45DtL0OLYwXc8mxSRToMPpoqKK1KgG'}]
Get user password=[{'password': '$2b$12$cyBil40ZhYOiweJbCMqI6.l45DtL0OLYwXc8mxSRToMPpoqKK1KgG'}]
Login user password=$2b$12$7f1O8UFFsQvlhutQdROi4eV0RYBJiZbWai5zBRqbkQ6v72l1Wpr7.
Production Test: Password attempt limit reached
Please enter the valid password, and try again.
```

Figure 6 - Incorrect Login Limit

We can also see the log of incorrect attempts by logging in as an admin user and accessing administrative controls.

This is seen from Figure 7.



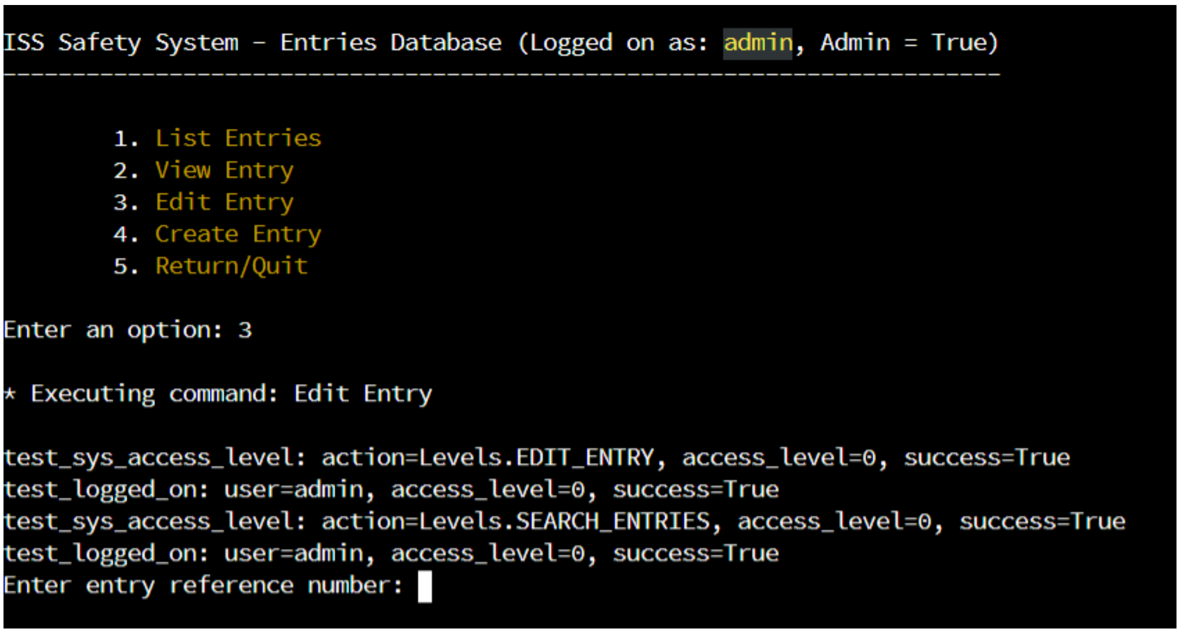
```
ISS System - Admin Management (Logged on as: admin, Admin = True)
-----------------------------------------------------------------


        1. Show Security Log
        2. Allocate User Token
        3. Show User Tokens
        4. Show Users
        5. Return/Quit

Enter an option: 1

* Executing command: Show Security Log

(Admin : 3)
```

Figure 7 - Incorrect Access Log

2.3 Access level

Within the system an authorisation class has been implemented. This class allows action to be limited dependant on the class of a user. An example of this would be that an admin can edit an entry, whereas a standard user cannot.

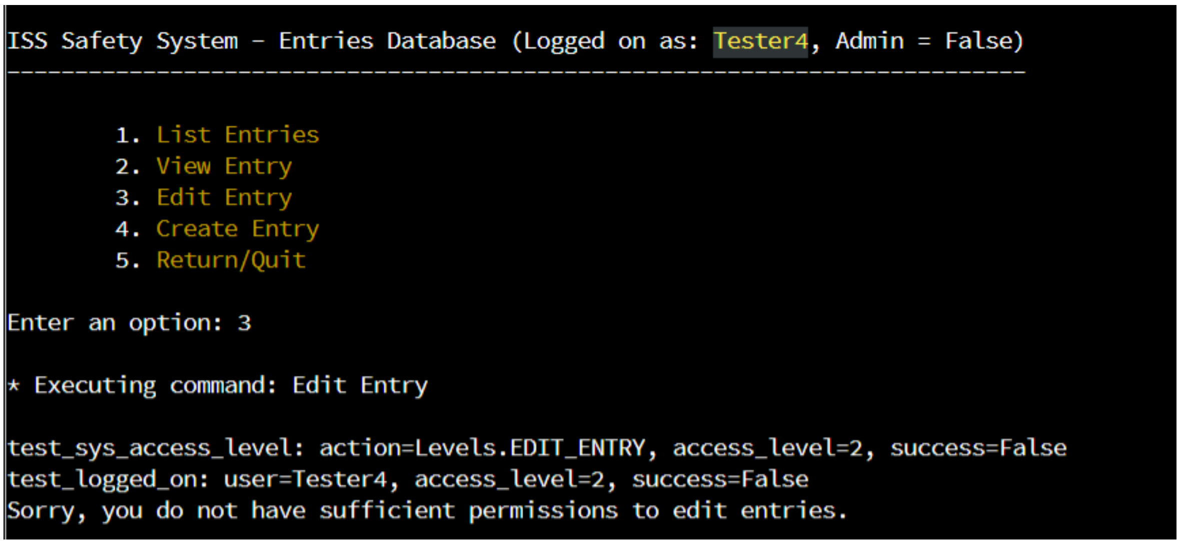
An example of an admin accessing the edit function is seen below from Figure 8.



```
ISS Safety System - Entries Database (Logged on as: admin, Admin = True)
------------------------------------------------------------------------

        1. List Entries
        2. View Entry
        3. Edit Entry
        4. Create Entry
        5. Return/Quit

Enter an option: 3

* Executing command: Edit Entry

test_sys_access_level: action=Levels.EDIT_ENTRY, access_level=0, success=True
test_logged_on: user=admin, access_level=0, success=True
test_sys_access_level: action=Levels.SEARCH_ENTRIES, access_level=0, success=True
test_logged_on: user=admin, access_level=0, success=True
Enter entry reference number:
```

Figure 8 - Authorisation admin using edit

It can also be shown that a standard user is prohibited from taking the same action. This can be seen from Figure 9.



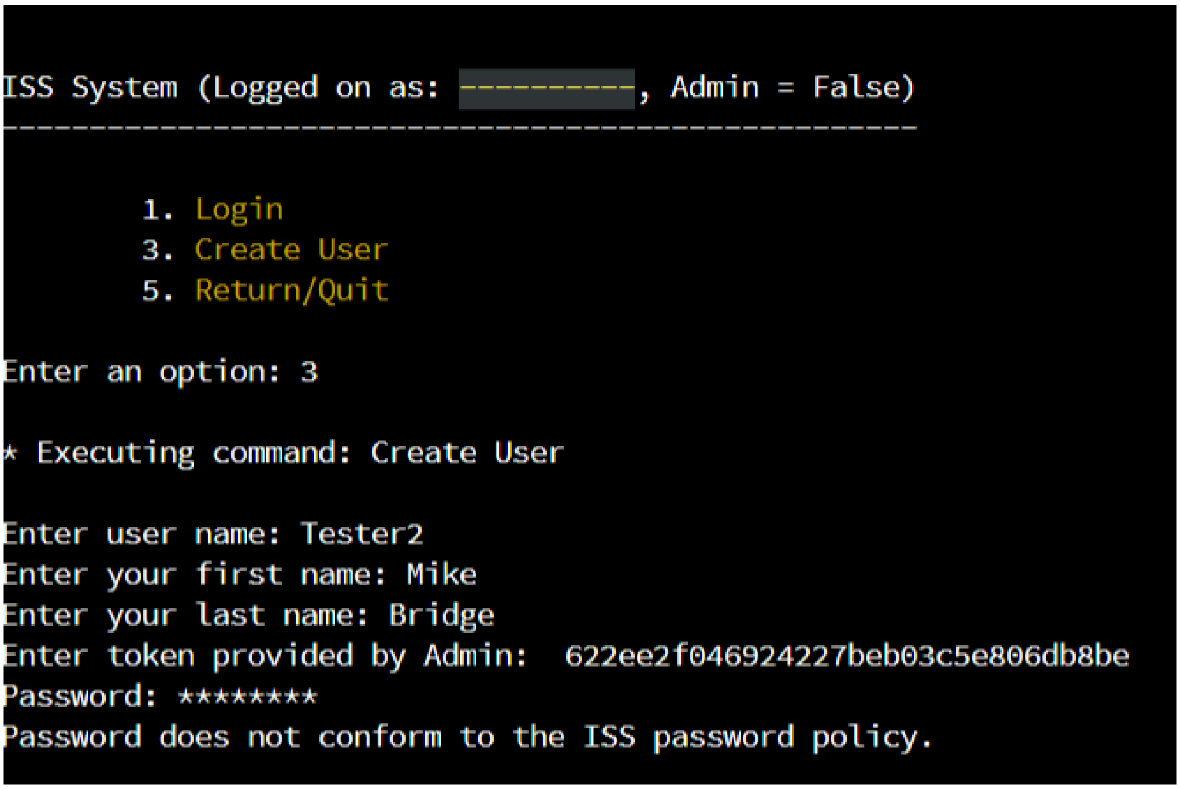
```
ISS Safety System - Entries Database (Logged on as: Tester4, Admin = False)
--------------------------------------------------------------------------

        1. List Entries
        2. View Entry
        3. Edit Entry
        4. Create Entry
        5. Return/Quit

Enter an option: 3

* Executing command: Edit Entry

test_sys_access_level: action=Levels.EDIT_ENTRY, access_level=2, success=False
test_logged_on: user=Tester4, access_level=2, success=False
Sorry, you do not have sufficient permissions to edit entries.
```

Figure 9- Authorisation Standard User Accessing Edit

2.4 *Password rules*

When a password is created withing the system it must conform to set rules. These rules are that it must contain one capital letter, one lower case letter, one number, one special symbol and be at least 7 characters long. These password rules have been implemented using a regular expression. We have also confirmed that the expression is not an “Evil Regex” (Van Der Merwe et al, 2017).

This can be tested by initially entering an incorrect password. For this example the phrase “password” is entered, which throws up an invalid password warning. This can be seen from Figure 10.



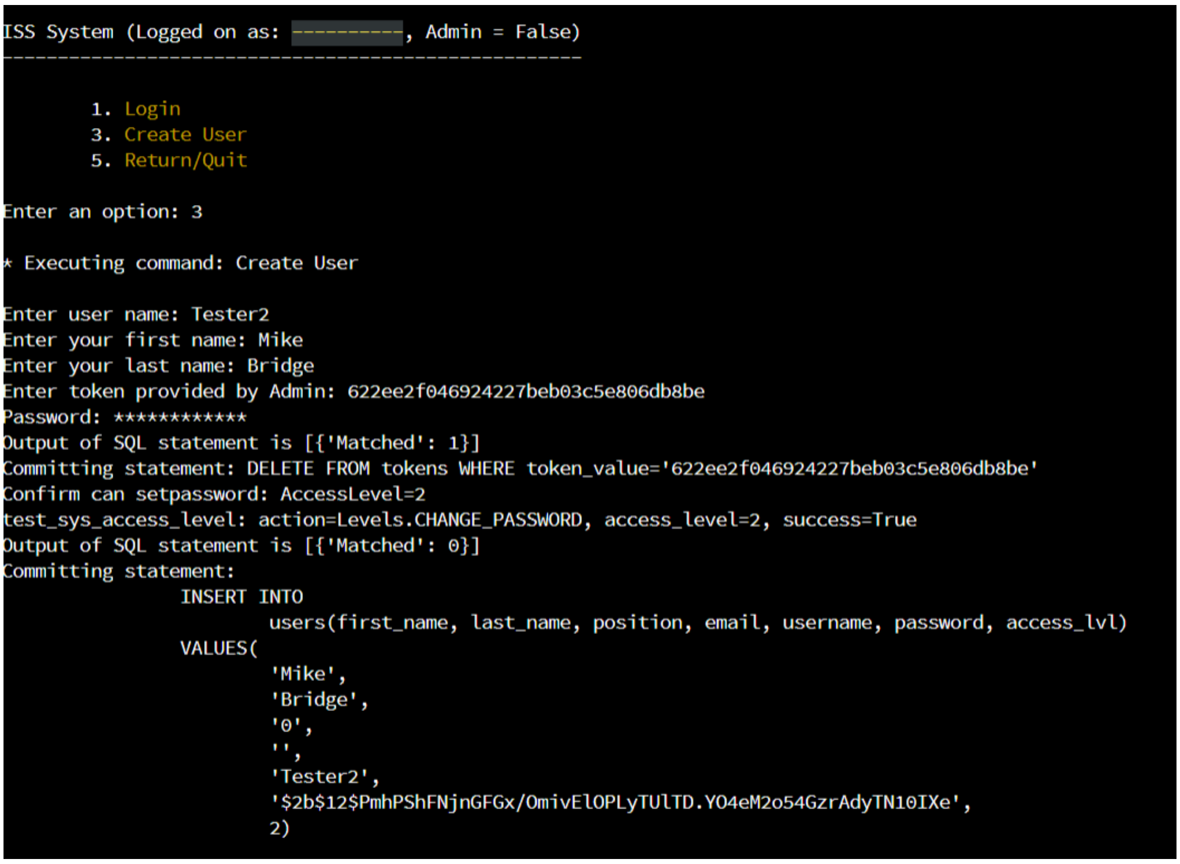
```
ISS System (Logged on as: ----------, Admin = False)
----------------------------------------------------

        1. Login
        3. Create User
        5. Return/Quit

Enter an option: 3

* Executing command: Create User

Enter user name: Tester2
Enter your first name: Mike
Enter your last name: Bridge
Enter token provided by Admin:  622ee2f046924227beb03c5e806db8be
Password: ********
Password does not conform to the ISS password policy.
```

Figure 10 - Password Does Not Meet Rules

Each rule has been tested thoroughly, but this simple demonstration will suffice for this document.

We can then pass a password that meets the correct format, this time “Pasword123!”, to ensure that it is accepted.

This is seen from Figure 11.



```
ISS System (Logged on as: ----------, Admin = False)
----------------------------------------------------

        1. Login
        3. Create User
        5. Return/Quit

Enter an option: 3

* Executing command: Create User

Enter user name: Tester2
Enter your first name: Mike
Enter your last name: Bridge
Enter token provided by Admin: 622ee2f046924227beb03c5e806db8be
Password: ************
Output of SQL statement is [{'Matched': 1}]
Committing statement: DELETE FROM tokens WHERE token_value='622ee2f046924227beb03c5e806db8be'
Confirm can setpassword: AccessLevel=2
test_sys_access_level: action=Levels.CHANGE_PASSWORD, access_level=2, success=True
Output of SQL statement is [{'Matched': 0}]
Committing statement:
                INSERT INTO
                        users(first_name, last_name, position, email, username, password, access_lvl)
                VALUES(
                        'Mike',
                        'Bridge',
                        '0',
                        '',
                        'Tester2',
                        '$2b$12$PmhPShFNjnGFGx/OmivElOPLyTUlTD.YO4eM2o54GzrAdyTN10IXe',
                        2)
```

Figure 11 - Correct Password Format

3. API

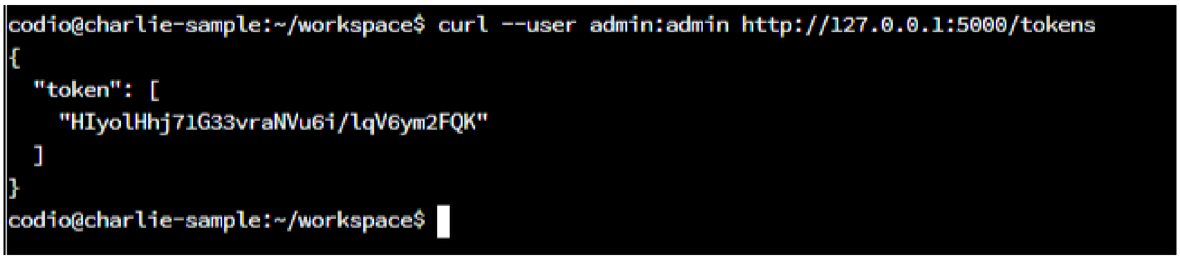
3.1 Session Token

The session token allows a user to generate a unique token to access the system without inputting their username and password each time. Each token has been set to last for a 1-hour duration and are revoked if used after the set time frame.

Initial code for this was taken from the mega flask tutorial (Ginberg, 2017). The API can be run with the command “python3 API/API.py” in the terminal.

3.1.1 Token generation testing

To ensure that the session token is working we initially generate a token.

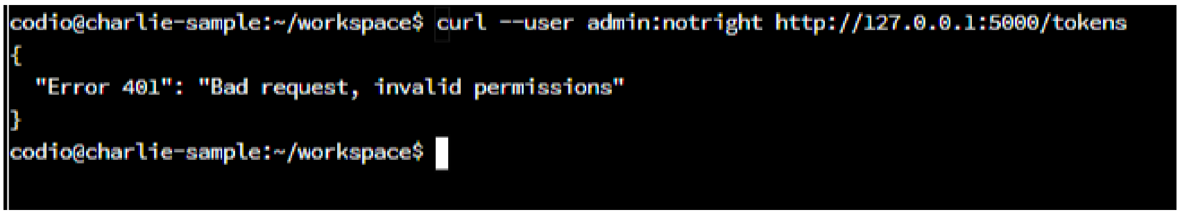


```
codio@charlie-sample:~/workspace$ curl --user admin:admin http://127.0.0.1:5000/tokens
{
  "token": [
    "HIyolHhj71G33vraNVu6i/lqV6ym2FQK"
  ]
}
codio@charlie-sample:~/workspace$
```

Figure 12 - Correct Token Generation

A token is called by calling the endpoint “/tokens”. Along with the endpoint a registered username and password are also parsed to the code. This is seen to function correctly from Figure 12. In this case the default username and password provided is “admin” and “admin”

We can also test that is correctly validating the input username and password by providing incorrect user details, such as an incorrect password. This can be seen from Figure 13.

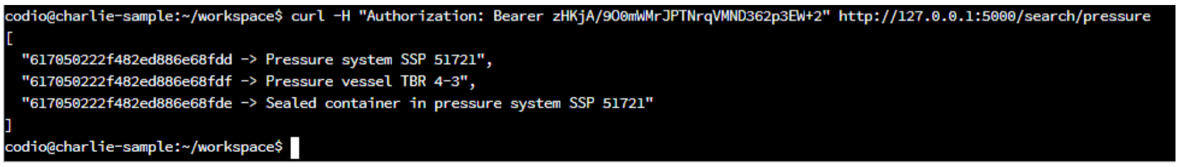


```
codio@charlie-sample:~/workspace$ curl --user admin:notright http://127.0.0.1:5000/tokens
{
  "Error 401": "Bad request, invalid permissions"
}
codio@charlie-sample:~/workspace$
```

Figure 13 - Token Generation - Invalid password

3.1.2 Functional testing

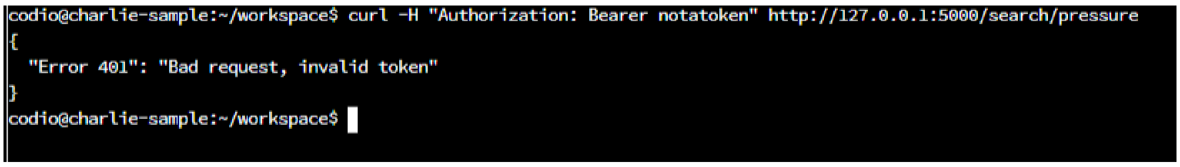
We can test that the token is working as intended by calling an endpoint that requires a valid token to return the intended output. For this test we will be using the search function set to the endpoint “"/search/<string:search_string>". The test with valid token is as seen from Figure 14.



```
codio@charlie-sample:~/workspace$ curl -H "Authorization: Bearer zHKjA/9O0mWMrJPTNrqVMND362p3EW+2" http://127.0.0.1:5000/search/pressure
[
  "617050222f482ed886e68fdd -> Pressure system SSP 51721",
  "617050222f482ed886e68fdf -> Pressure vessel TBR 4-3",
  "617050222f482ed886e68fde -> Sealed container in pressure system SSP 51721"
]
codio@charlie-sample:~/workspace$
```

Figure 14 - Correct Token Validation

We can also test its functionality, by inputting an invalid token. This is seen from Figure 15.

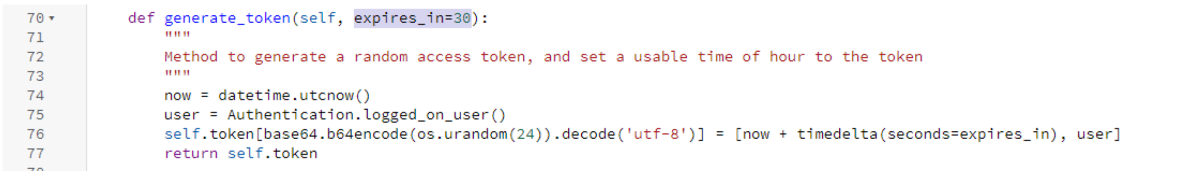


```
codio@charlie-sample:~/workspace$ curl -H "Authorization: Bearer notatoken" http://127.0.0.1:5000/search/pressure
{
  "Error 401": "Bad request, invalid token"
}
codio@charlie-sample:~/workspace$
```

Figure 15 - Incorrect Token Validation

3.1.3 Expiry testing

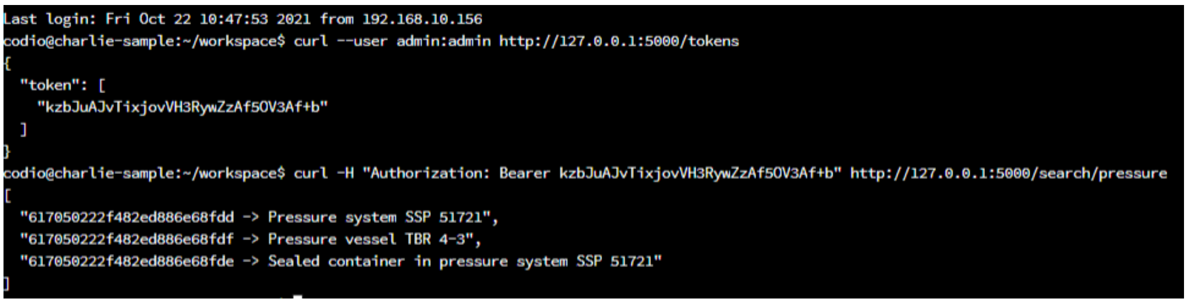
By default, each token is set to expire after 3600 seconds (1 hour), but for the testing this has been altered to 30 seconds. This is set on line 70 as seen from line 70 of Figure 16



```
70      def generate_token(self, expires_in=30):
71          """
72          Method to generate a random access token, and set a usable time of hour to the token
73          """
74          now = datetime.utcnow()
75          user = Authentication.logged_on_user()
76          self.token[base64.b64encode(os.urandom(24)).decode('utf-8')] = [now + timedelta(seconds=expires_in), user]
77          return self.token
```

Figure 16 - Token Generation Code

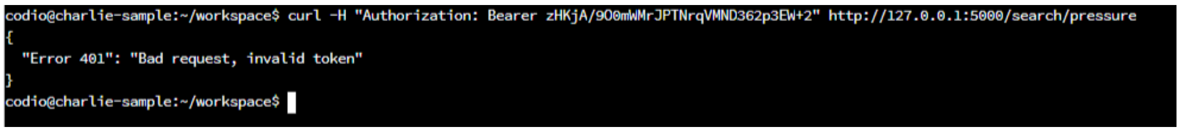
Initially we generate a token as previously. Within the 30 second time frame, we again call the search end point to prove that the token is valid and working. This is seen in Figure 17.



```
Last login: Fri Oct 22 10:47:53 2021 from 192.168.10.156
codio@charlie-sample:~/workspace$ curl --user admin:admin http://127.0.0.1:5000/tokens
{
  "token": [
    "kzbJuAJvTixjovVH3RywZzAf5OV3Af+b"
  ]
}
codio@charlie-sample:~/workspace$ curl -H "Authorization: Bearer kzbJuAJvTixjovVH3RywZzAf5OV3Af+b" http://127.0.0.1:5000/search/pressure
[
  "617050222f482ed886e68fdd -> Pressure system SSP 51721",
  "617050222f482ed886e68fdf -> Pressure vessel TBR 4-3",
  "617050222f482ed886e68fde -> Sealed container in pressure system SSP 51721"
]
```

Figure 17 - Token Validity Expiry Test - Valid

After waiting 30 seconds we try to use the same token again. This time we receive a “401” error as the token is no longer valid and has been revoked due to it being outside the expiry time. This is seen in Figure 18



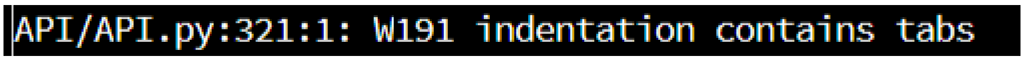
```
codio@charlie-sample:~/workspace$ curl -H "Authorization: Bearer zHKjA/9O0mWMrJPTNrqVMND362p3EW+2" http://127.0.0.1:5000/search/pressure
{
  "Error 401": "Bad request, invalid token"
}
codio@charlie-sample:~/workspace$
```

Figure 18 - Token Validity Test - Expired Token

4. Linters

Throughout testing linters have been applied to confirm that the code has met PEP8 guidelines for coding format. The package used for this purpose was flake8. Flake8 also allows us the leverage the ability to test for cyclomatic complexity. Although there are many academic arguments for and against cyclomatic complexity, it gave us a relative guideline to the complexity of our code.

It should be noted that we allowed some stylistic errors to remain from the Flake 8 output, as we believed our own stylistic choices enabled us to develop more clearly readable code. One example of an existing error is W191, as is seen from Figure 19 - Example Flake8 Output.



```
API/API.py:321:1: W191 indentation contains tabs
```

Figure 19 - Example Flake8 Output

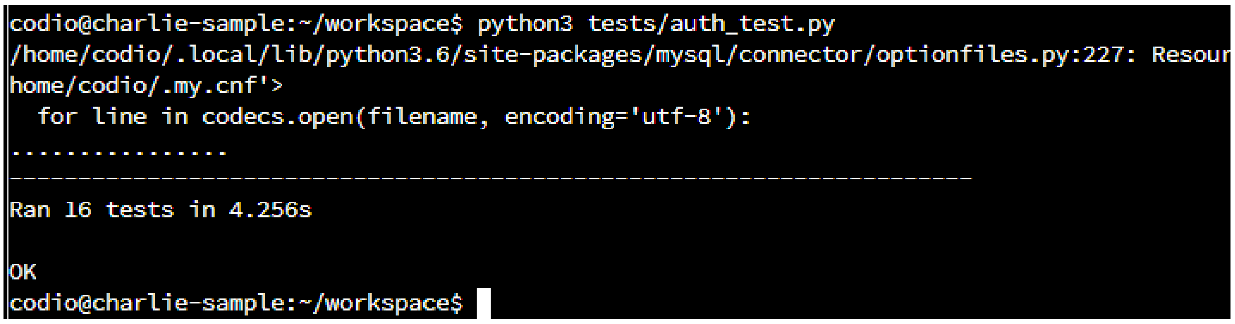
The w191 warning notifies that tab are used instead of spaces. As this has a

negligible effect on the code and how it is displayed we decided our time would be better spent elsewhere than correcting these errors.

5. Whitebox Testing

To assist with the internal testing of the software classes and components we developed a set of unit tests to provide a form of Whitebox testing. These tests can be found in the file “tests” and is composed of “auth_tests.py” and “user_repor_tests.py”. These set of tests help to confirm the individual components of both the user database and authentication are working as intended.

The output from the automated tests can be seen from Figure 20 - Automated Tests.



```
codio@charlie-sample:~/workspace$ python3 tests/auth_test.py
/home/codio/.local/lib/python3.6/site-packages/mysql/connector/optionfiles.py:227: Resour
home/codio/.my.cnf'>
  for line in codecs.open(filename, encoding='utf-8'):
................
----------------------------------------------------------------------
Ran 16 tests in 4.256s

OK
codio@charlie-sample:~/workspace$
```

Figure 20 - Automated Tests

As there are no errors risen it shows that the individual components are working correctly from the internal testing.

6. References

Umar, M. (2019) Comprehensive study of software testing: Categories, levels, techniques, and types. *International Journal of Advance Research, Ideas and Innovations in Technology* 5(6): 32-40.

Van Rossum, G., Warsaw, B. and Coghlan, N., 2001. PEP 8: style guide for Python code. Python. org, 1565.

Mitre (2017) Weaknesses in OWASP Top Ten.

Van Der Merwe, B., Weideman, N. and Berglund, M., 2017. Turning evil regexes harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists* (pp. 1-10).

Ginberg, M. (2017) The Flask Mega Tutorial