Question 1

The Registration.name() function is pasted below.

```
/*
name()

Given a student name in a string "Firstname Lastname", this
function consults the StudentDatabaseConnection database, and
returns the student ID provided by that database.

If the argument is an invalid student number, an InvalidIDNumberException
is thrown.

Precondition: If the database is not connected, a DatabaseNotConnected
exception is thrown.

Post-condition:  This function will never return null to the caller.
 */


public String name(String idNumber) {
    //Throw a database not connected exception if database is not connected
    // add precondition design-by-contract code here.
    if (!database.isConnected()) {
        throw new DatabaseNotConnected();
    }
    if (!isValidIDNumber(idNumber)) {
        throw new InvalidIDNumberException();
    }
    String output = database.nameFromIDNumber(idNumber);
    System.out.println(output);
    // add post-condition design-by-contract code here.

    assert output != null;
    return output;
}
```

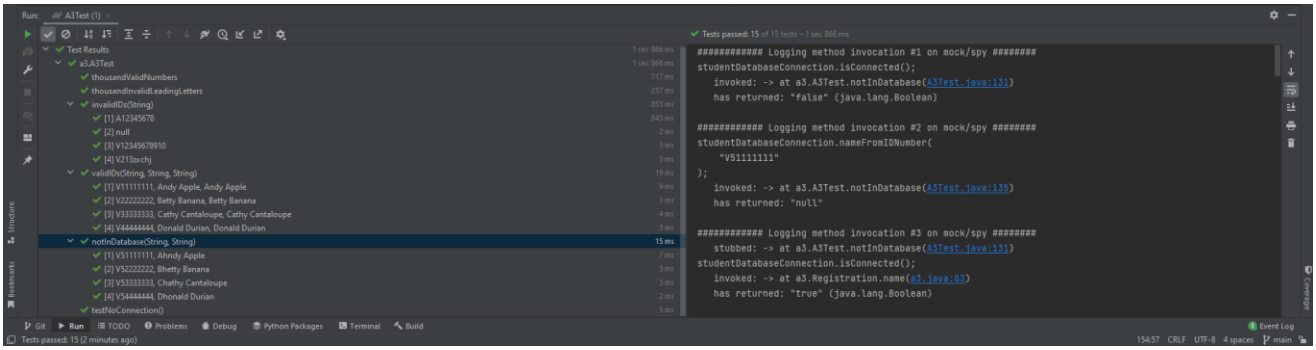
The tests for assignment 3 part 1 are pasted below.

```java
package a3;

import static org.junit.jupiter.api.Assertions.*;
import static org.assertj.core.api.Assertions.*;
import static org.mockito.ArgumentMatchers.*;
import static org.mockito.Mockito.*;

import net.jqwik.api.*;
import net.jqwik.api.constraints.*;

import java.io.IOException;
import java.util.stream.*;

import org.junit.jupiter.api.*;
import org.junit.jupiter.params.*;
import org.junit.jupiter.params.provider.*;

class A3Test {
    private StudentDatabaseConnection connection;
    private Registration reg;
    private String[] idNums;
    private String[] names;

    /**Question 1 part 1
     * The code under test is the two functions in the registration class:
     * (name(String idNUmber) and isValidIDNumber(String idNumber) )
     */

    @BeforeEach
    void setup() {
        //Create a mock connection and create a registration object with mocked connection
        connection = mock(StudentDatabaseConnection.class, withSettings().verboseLogging());
        reg = new Registration(connection);
        idNums = new String[]{"V11111111", "V22222222", "V33333333", "V44444444"};
        names = new String[] {"Andy Apple", "Betty Banana", "Cathy Cantaloupe", "Donald Durian"};



    }

    // Implement your tests below.  Feel free to modify the function signatures
    // as necessary - use parameterized testing as you see fit.

    // test for correct behaviour when the database is not connected.

    @Test
    void testNoConnection() {
        //Use test double to ensure that no connection is established when registration is called
        for (String input: idNums) {
            when(connection.isConnected()).thenReturn(false);
            //No need for parameterized testing here as database isn't even connected to check for multiple names
            //Used loop to make sure that entire given input domain was covered
            //Make sure correct exception is thrown
            assertThrows(DatabaseNotConnected.class, () -> { reg.name(input); } );


        }
        //Verify connection is only checked once per each id
        verify(connection, times(idNums.length)).isConnected();

    }
```

```
63      // test for correct behaviour when the database is connected, but invalidly
64      //    formatted IDs are submitted.
65
66      //Not interested in how invalidIdNumber works ( or if it does in the first place)
67      //However we can't mock static methods with standard mockito
68      //Instead we will run the test with the input being things we know are invalid ids
69      //They will not start with v, some may be null, some may have a length other than 9
70      //
71      @ParameterizedTest
72      @CsvSource({
73              "'A12345678', Doesn't start with a V but is 9 chars and has rest numbers",
74              "null, is null",
75              "'V12345678910', starts with a v and isn't null but is over 8 digits for number portion",
76              "'V213zxchj', has 8 digits and starts with a v but contains non numbers in numbers portion"
77
78
79      })
80      void invalidIDs(String input) {
81          when(connection.isConnected()).thenReturn(true);
82          System.out.println(input);
83
84          assertThrows(InvalidIDNumberException.class, ()->{reg.name(input);});
85
86          //Verify connection is only checked once per each id
87          verify(connection, times( wantedNumberOfInvocations: 1)).isConnected();
88          //verify(reg, times(1)).name(input);
89
90      }
91
92      // test for correct behaviour when the database is connected and students in
93      //     the database are searched for using the correct IDs.
94
95      @ParameterizedTest
96      @CsvSource({
97              "'V11111111', 'Andy Apple',      'Andy Apple' ",
98              "'V22222222', 'Betty Banana',    'Betty Banana'",
99              "'V33333333', 'Cathy Cantaloupe','Cathy Cantaloupe'",
100             "'V44444444', 'Donald Durian',   'Donald Durian'"
101     })
102     void validIDs(String input, String name, String expected) {
103         //Assume nameFromIDNumber will give us the name in database
104         when(connection.nameFromIDNumber(input)).thenReturn(name);
105         //Assume connection is working
106
107         when(connection.isConnected()).thenReturn(true);
108         assertEquals(reg.name(input), expected);
109         //Make sure only 1 call is made for each input
110         verify(connection, times( wantedNumberOfInvocations: 1)).isConnected();
111         verify(connection, times( wantedNumberOfInvocations: 1)).nameFromIDNumber(input);
112
113     }
```

```
    // test for correct behaviour when the database is connected and IDs are provided but
    // although they are the correct format (correct length, etc), there is no corresponding
    // student in the database.

    @ParameterizedTest
    @CsvSource({
            "'V51111111', 'Ahndy Apple' ",
            "'V52222222', 'Bhetty Banana'",
            "'V53333333', 'Chathy Cantaloupe'",
            "'V54444444', 'Dhonald Durian'"
    })
    void notInDatabase(String input, String name) {



        //Assume connection is working
        when(connection.isConnected()).thenReturn(true);

        //If the ID number does not correspond to any student in the database,
        // Then a StudentNotFound exception is raised.
        when(connection.nameFromIDNumber(input)).thenThrow(new StudentNotFoundException());

        assertThrows(StudentNotFoundException.class, ()->{reg.name(input);});

        //Check methods for interface are only called once per input
        verify(connection, times( wantedNumberOfInvocations: 1)).isConnected();
        verify(connection,times( wantedNumberOfInvocations: 1)).nameFromIDNumber(input);
    }

    // Test whether isValidIDNumber() accepts 1000 properly-formatted ID numbers

    //@IntRange(max = 1000) int me
    @Property

    void thousandValidNumbers(@ForAll("createNumbersWithinRange") int inputBeforeV) {

        String input = "V"+inputBeforeV;

        System.out.println(input);
        assertTrue(Registration.isValidIDNumber(input));
        //We care about numbers between 10000000 and 99999999 as they are 8 digits and will have V slapped in front of

    }

    // Test whether isValidIDNumber() accepts 1000 ID numbers which are correctly formatted
    // except that their first letter is a lowercase letter rather than an uppercase V.
    @Property
    void thousandInvalidLeadingLetters(@ForAll("createNumbersWithinRange") int inputBeforeLetter, @ForAll @LowerChars c

        String input = ""+the_char+inputBeforeLetter;
        System.out.println(input);
        assertFalse(Registration.isValidIDNumber(input));
    }

    //Creates numbers between the acceptable digit range for a student number (8 digit number)
    @Provide
    Arbitrary<Integer> createNumbersWithinRange(){

        return Arbitraries.integers().filter(n -> n>=10000000 && n<=99999999);
    }

}
```

The results for the tests are pasted here.



Question 2

1.) Five high alert vulnerabilities are.
   - http://testasp.vulnweb.com/
     i. Path Traversal
     ii. Reflected Cross Site Scripting
     iii. SQL Injection
     iv. External Redirect

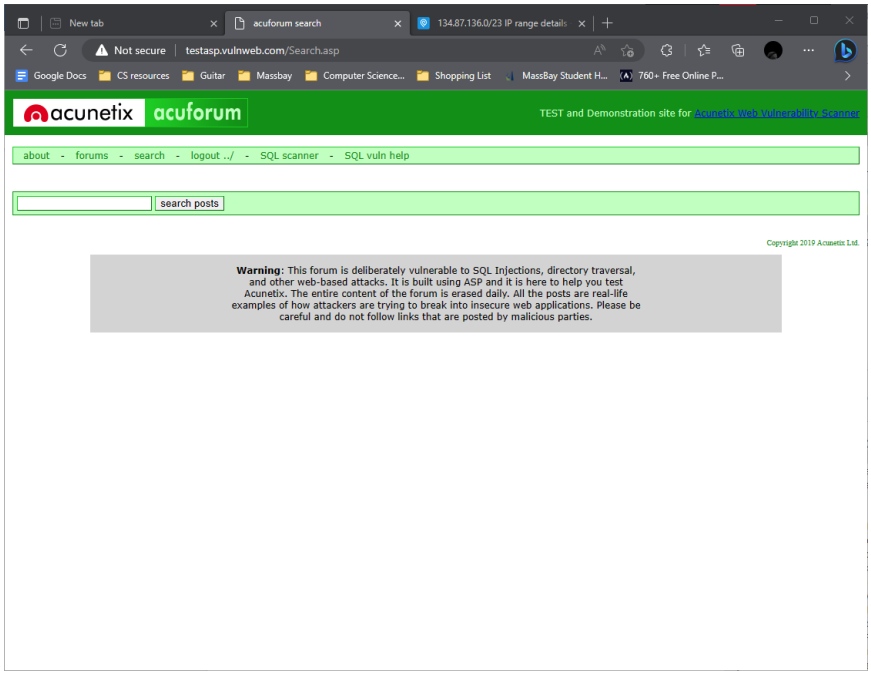
2.) The five vulnerabilities have the following description and were found with OWASP ZAP 2.12.0
   - Path Traversal – an attack where the attacker has access to files and directories outside of the web document root directory. Per OWASP ZAP this attack is often carried out by using "../" which changes the current directory to the parent directory to access files anywhere on the web server.
   - Reflected Cross Site Scripting- Attack that has attacker supplied code entered into a browser to be executed.
   - SQL Injection – Attack where attacker enters an SQL query as input to run on website's database.
   - External Redirect- External redirection itself is not an attack. However, an attacker can use social engineering to make users believe that they are at a site that is trusted when in reality it is an entirely different site. This different site can pose as another site and prompt the user to login with their account details at which point, they are at risk of being stolen
   -

3.) The vulnerabilities are critical weaknesses and can be exploited in the following ways.
   - Path Traversal- Is a critical weakness because it allows for files on the web server to be revealed. It can be executed by using a special sequence of characters used to refer to a parent directory "../"
   - Reflected Cross Site Scripting
   - SQL injection- An attacker can use SQL injection to reveal the usernames and passwords if they are stored on a database used by the website in plain text
   - External Redirect- An external redirect may lead an unsuspecting user to a phishing site where they are tricked into revealing their username and password to an attacker. This

becomes more dangerous if a user uses the same login credentials for other sites or all the logins of the users for the site follow a pattern such as firstname_lastname for the username, and a user's year of birth for the password.

4.) The vulnerabilities can be exploited in the following ways

- http://testasp.vulnweb.com/
    i. Path Traversal- Enter ../ Into registration fields to get to a higher directory and to sign in to an account that you didn't create with a valid email.



    ii. Reflected Cross Site Scripting
    iii. SQL Injection – Enter SQL
    iv. External Redirect

5.) To fix the issues the following strategies are recommended:

- Path Traversal- We can assume that all input is malicious and allow only certain inputs to be used that agree with the requirements of the website. We most likely don't need to allow the specific sequence "../" on our website. So we will validate inputs to make sure that "../" cannot be entered
- Reflected Cross Site Scripting- To prevent cross site scripting I would perform input validation to ensure that no attacker supplied code is entered. I would also recommend doing security checks on both the client side and server side.
- SQL Injection- To prevent this I would have a list of allowed input and reject any input with disallowed characters such as "="
- External Redirect- To fight against this I would recommend a list of allowed redirects on the site in question. I would also include a warning for when users are being redirected off site as seen on many 3rd party shopping sites