Term-Long project: Final Report

Topic: Excessive Data Exposure

COSC 4P50 - Introduction to Cyber Security

December 9, 2022

Group members:

- Yelyzaveta Denysova
- Shahrear Chowdhury
- Dhrumil Rajesh Shah
- Shubham Nileshbhai Amrelia

Brock University

1812 Sir Isaac Brock Way,

St. Catharines, ON L2S 3A1

Table Of Contents

Introduction

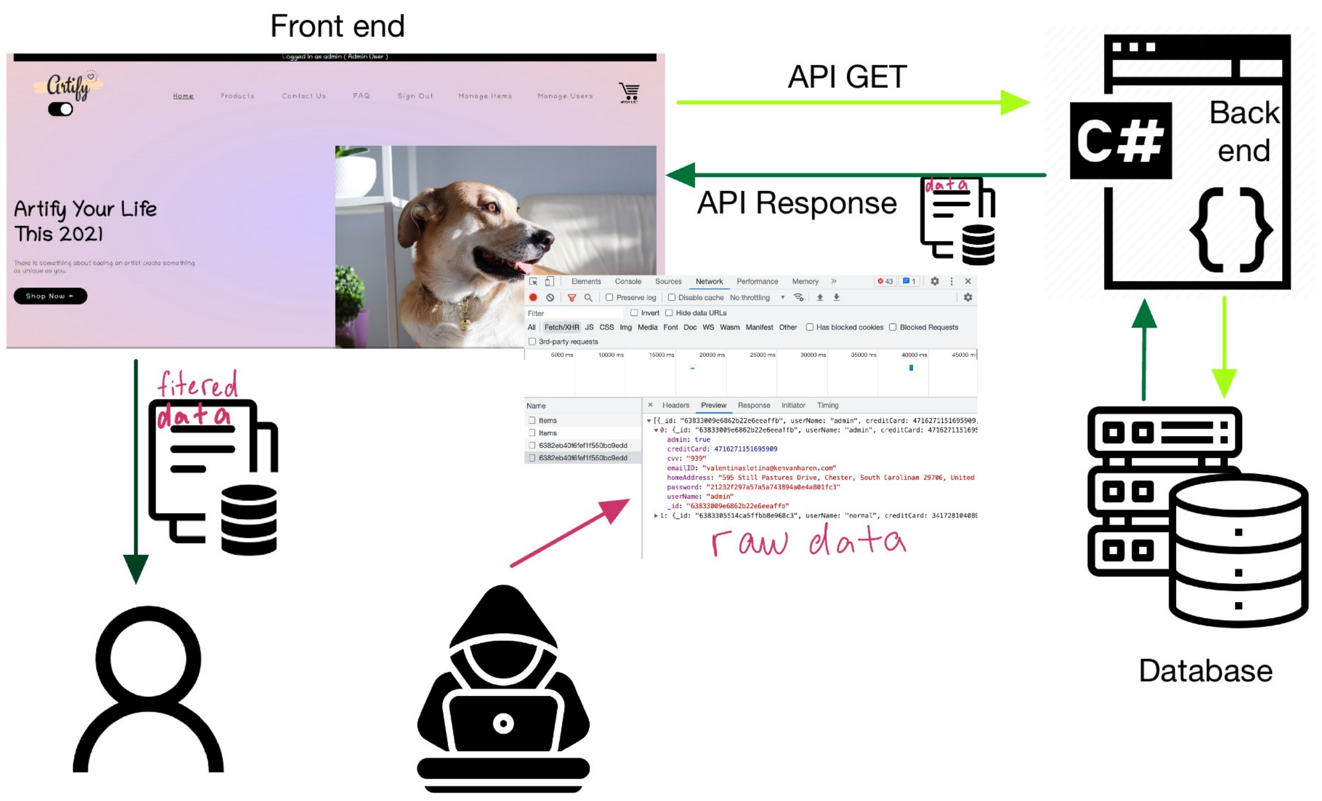
According to Open Web Application Security Project® (OWASP) in the report "Top 10 Web Application Security Risks" for 2021 cryptographic failures previously known as sensitive data exposure takes the second position of the list. Excessive data exposure is a web security vulnerability that occurs when the front-end of the application receives data (usually from API GET requests) and does the filtering to show to users only the data they require. The attacker can exploit this by reviewing the raw data from the server's response and exposing sensitive data.

Why does this vulnerability have such a high ranking? The OWASP's report is intended to increase the awareness of the potential security risks and draw developers attention. This specific security flaw often slips out of the developers mind as they are more focused on delivering data to the front-end. There is also a possible miscommunication between developers working on the front-end and back-end of an application. Workers creating the API responses might rely on the filtering of the front-end, taking no regard to the sensitivity of data. However, many websites store a lot of sensitive data such as personal information of clients. Such data is not intended to be viewed by other people and can deal lots of damage when exposed. What is the solution? Again considering a high ranking of popularity of this security risk it is not difficult to guess that so far there is no perfect solution to prevent or detect excessive data. If you were to search for it online most of the suggestions are to be aware of the sensitive data or to triple check the server responses. There is one very helpful idea hidden in the name of this risk: cryptography. Ensuring that all data being sent to the front-end is encrypted is a great way to prevent the issue. However, what if the website application is old and enabling encryption is troublesome?

The task of this project is to look into this issue and develop a solution that could help developers prevent sensitive data exposure without changing the whole infrastructure. The idea is to have a program that could collect raw data from API response and analyse for sensitive data exposure.

Goal

The goal of the project is to display the vulnerability and demonstrate the efficiency of the proposed defense. There were few intermediate goals during multiple stages of development. Firstly, the front-end was created for the online store. Created a back-end that delivers data from the database and planted a security flaw in the code. The back-end API is intended to be connected to front-end instead of straight connection from front-end to database as it is right now. The API response that returns sensitive data about customers was analyzed to show vulnerability and a test was developed to propose as a defensive technique. Finally, examine the effectiveness of the defense. Next section describes all the parts of implementation in detail. All codes are included in the submission folder in corresponding subfolders.



```
▼ [{_id: "63833009e6862b22e6eeaffb", userName: "admin", creditCard: 4716271151695909,
  ▼ 0: {_id: "63833009e6862b22e6eeaffb", userName: "admin", creditCard: 471627115169
      admin: true
      creditCard: 4716271151695909
      cvv: "939"
      emailID: "valentinaslotina@kenvanharen.com"
      homeAddress: "595 Still Pastures Drive, Chester, South Carolinam 29706, United
      password: "21232f297a57a5a743894a0e4a801fc3"
      userName: "admin"
      _id: "63833009e6862b22e6eeaffb"
  ▶ 1: {_id: "6383305514ca5ffbb8e968c3", userName: "normal", creditCard: 34172810408
```

Fig. 1: Web Application Flow chart

Experimental set-up:

Front End

The front-end is built using HTML, CSS, JS, and Python.

Primary pages of the website includes:

1. A product listing page which displays the entirety of all products
2. An individual product page which displays any requested product
3. A wish list page which allows users to view and remove items added to their personal wish list
4. A user administration page which allows for adding and removing users
5. A product administration page which allows for adding and removing products
6. A 'sign up' page which allows the creation of new users accounts
7. A login mechanism which allows different parties accessing the website to authenticate themselves

Administering users and products is only performed by logged-in admin users. When logging in, any client-side wish list is merged with the server-side list.

Specific page-wise functionalities include:

Home:

1. Toggle between themes which remain the same across pages
2. Clicking Shop Now leads to Products page
3. Clicking anything else clickable also leads to Products page

Products:

1. Without Logging in, you are able to add to Wishlist from Products page with a notification of successful add
2. Clicking the view button, allows the user to view each product on an individual page
3. Ability to search for an item using tags or categories. For example, try any of the words below, not case-sensitive:
    a. Silver
    b. Gold
    c. Resin
    d. Green
    e. Heart
    f. Double
    g. Bone … etc.
4. After typing a word, more than 3 characters in length, and hitting enter, user is able to view all products matching the tag in a drop-down menu
5. Clicking any of these products in the dropdown, leads to individual product page

Individual Product Page:

1. One page to show all items using their ID from db on the same page
2. Ability to add to wish list with a notification of successful add
3. Clicking back takes you to products page

Contact Us:

1. For aesthetic purposes, no real function to this page

FAQ:

1. References to the resources used

Sign Out:

1. Whether logged in as a normal user or admin, it will sign the person out if not logged in, this button will not appear Log In:

1. When on the login page, use is able to login as admin or normal user using the password and user provided for testing purposes

2. User also has the option to sign up as a normal user only
3. Existing usernames cannot be used to sign in
4. Once logged in as admin, user is able to manage items: add new and remove existing with pictures
5. Once logged in as admin, user is able to manage other users: add new ones with either admin privileges or normal privileges and also delete existing users
6. Login status is displayed at the top of the page.

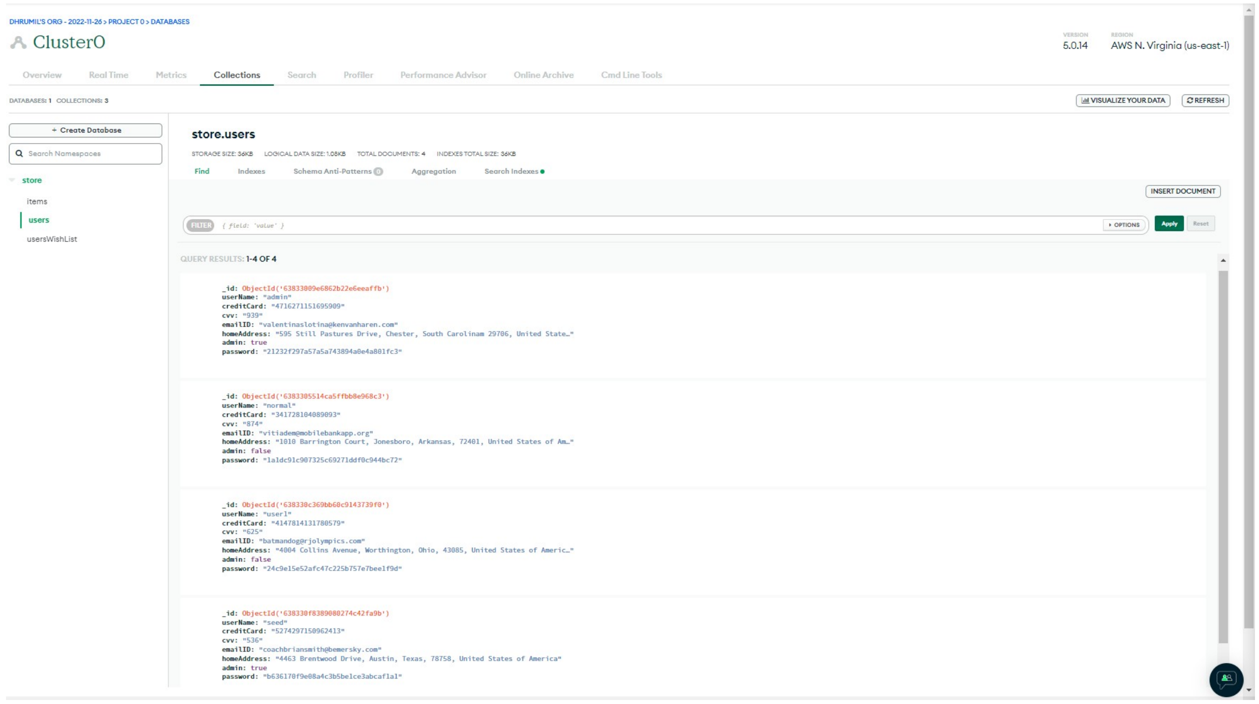
<u>Wish List:</u>

1. Once in the wish list, users whether logged in as admin, normal or not logged in, are able to delete any item in their wish list. While admin and normal users have actual saved wish lists created on the database, the user who is not logged in also as a wish list saved locally on their browser
2. Ability to print the wish list which is the wish list displayed as a table format with picture of item, name of item and price of item
3. After clicking print again, the print dialog box pops up
4. After clicking the share wish list, a custom wishlist page is formed of the specific user's cart with a link that is shareable to it. This page doesn't compromise the security of the user.

Database

<u>Type</u>: NoSQL

<u>Collections</u>: items, users, usersWishList



```
_id: ObjectId('63833809e6862b22e6eeaffb')
userName: "admin"
creditCard: "4716271151695909"
cvv: "939"
emailID: "valentinaslotina@kenvanharen.com"
homeAddress: "595 Still Pastures Drive, Chester, South Carolinam 29706, United State…"
admin: true
password: "21232f297a57a5a743894a0e4a801fc3"

_id: ObjectId('6383305514ca5ffbb8e968c3')
userName: "normal"
creditCard: "341728104089093"
cvv: "874"
emailID: "vitiadem@mobilebankapp.org"
homeAddress: "1010 Barrington Court, Jonesboro, Arkansas, 72401, United States of Am…"
admin: false
password: "1a1dc91c907325c69271ddf0c944bc72"

_id: ObjectId('638330c369bb60c9143739f0')
userName: "user1"
creditCard: "4147814131780579"
cvv: "625"
emailID: "batmandog@rjolympics.com"
homeAddress: "4004 Collins Avenue, Worthington, Ohio, 43085, United States of Americ…"
admin: false
password: "24c9e15e52afc47c225b757e7bee1f9d"

_id: ObjectId('638330f8389080274c42fa9b')
userName: "seed"
creditCard: "5274297150962413"
cvv: "536"
emailID: "coachbriansmith@bemersky.com"
homeAddress: "4463 Brentwood Drive, Austin, Texas, 78758, United States of America"
admin: true
password: "b636170f9e08a4c3b5be1ce3abcaf1a1"
```

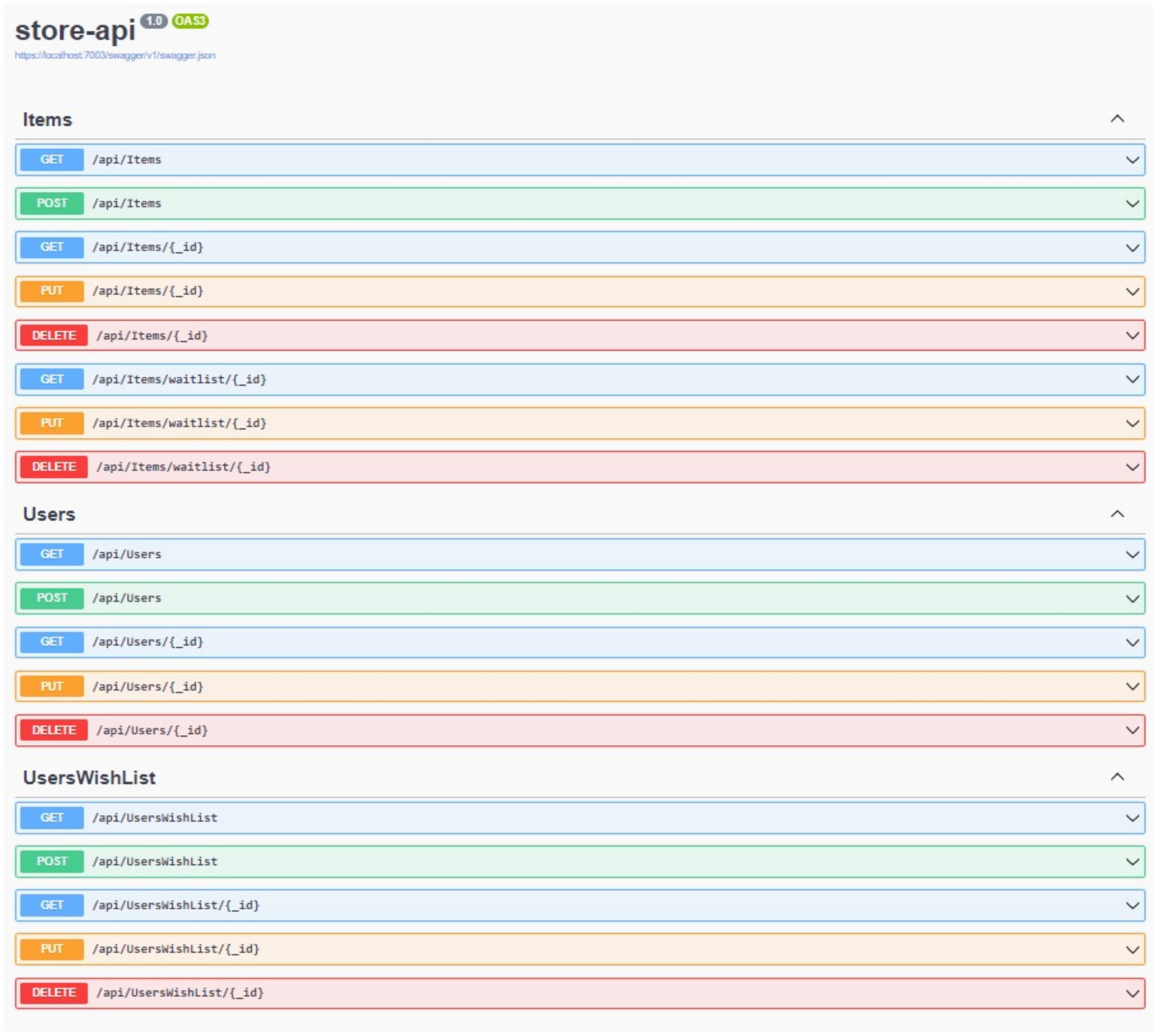
Fig. 2: Database

Backend - API

Requirements: .NET web API, C#, MongoDB(Database), Swagger UI, Visual Studio, URL(local) - https://localhost:7003.

Note: StoreService.cs in Services provides all the asynchronous methods for database operations (CRUD) and all Controller.cs in Controllers folder contains routes (GET/POST/PUT/DELETE) to use these database operation methods to different collections (Items/Users/UsersWishList).

API Endpoints in Swagger Visualization:


store-api 1.0 OAS3
https://localhost:7003/swagger/v1/swagger.json

Items
GET /api/Items
POST /api/Items
GET /api/Items/{_id}
PUT /api/Items/{_id}
DELETE /api/Items/{_id}
GET /api/Items/waitlist/{_id}
PUT /api/Items/waitlist/{_id}
DELETE /api/Items/waitlist/{_id}

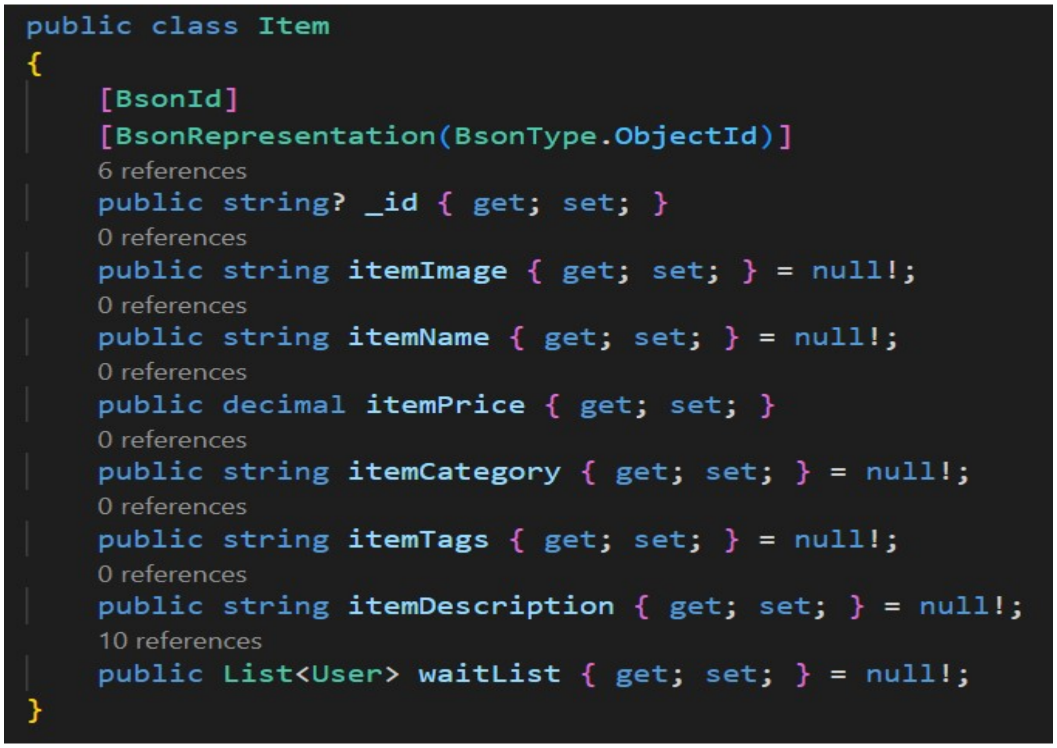
Users
GET /api/Users
POST /api/Users
GET /api/Users/{_id}
PUT /api/Users/{_id}
DELETE /api/Users/{_id}

UsersWishList
GET /api/UsersWishList
POST /api/UsersWishList
GET /api/UsersWishList/{_id}
PUT /api/UsersWishList/{_id}
DELETE /api/UsersWishList/{_id}

Fig. 3: API Endpoints

Database Models:

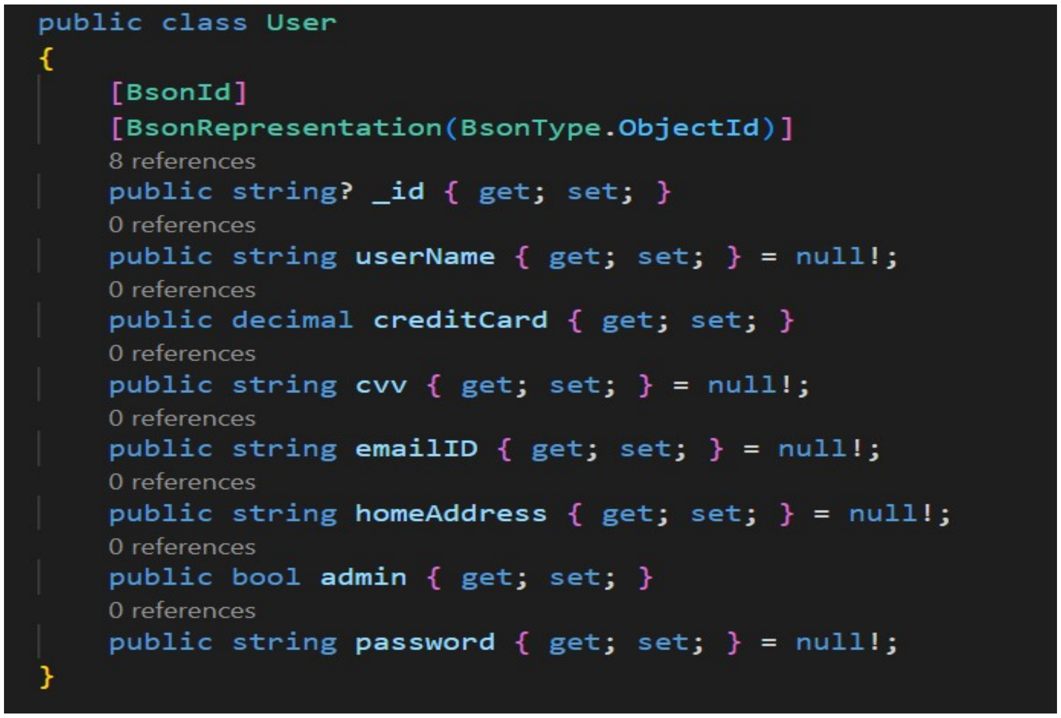
1. **Item:**



```
public class Item
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    6 references
    public string? _id { get; set; }
    0 references
    public string itemImage { get; set; } = null!;
    0 references
    public string itemName { get; set; } = null!;
    0 references
    public decimal itemPrice { get; set; }
    0 references
    public string itemCategory { get; set; } = null!;
    0 references
    public string itemTags { get; set; } = null!;
    0 references
    public string itemDescription { get; set; } = null!;
    10 references
    public List<User> waitList { get; set; } = null!;
}
```

Fig. 4: Item Model

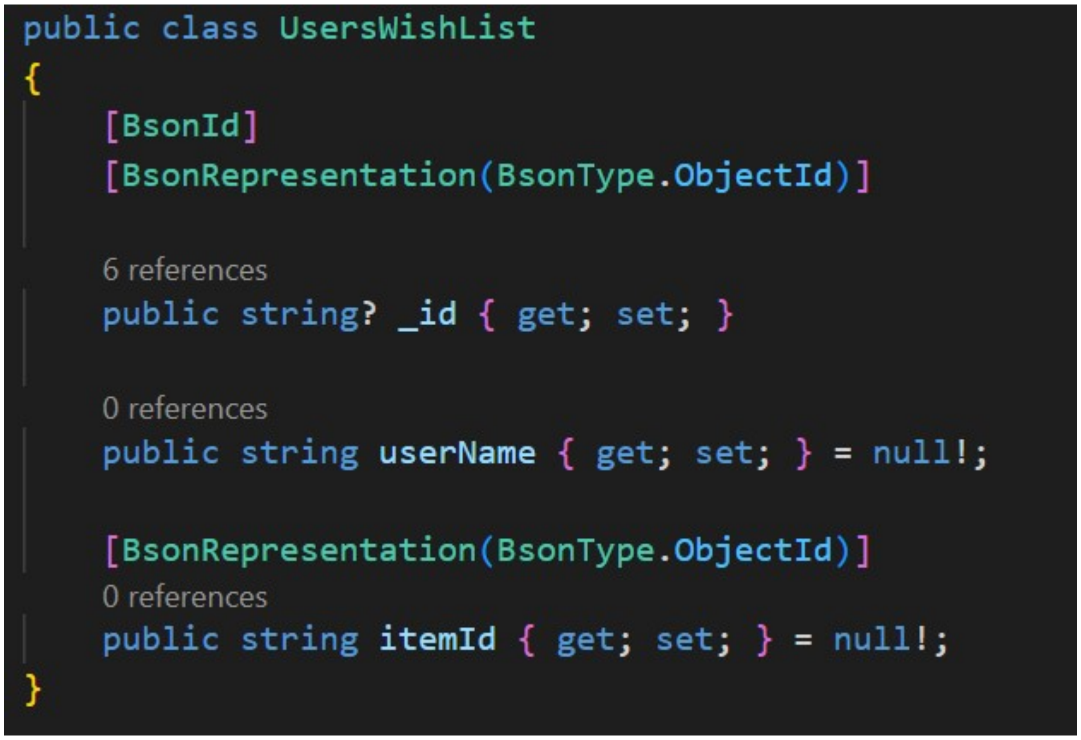
2. **User:**



```
public class User
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    8 references
    public string? _id { get; set; }
    0 references
    public string userName { get; set; } = null!;
    0 references
    public decimal creditCard { get; set; }
    0 references
    public string cvv { get; set; } = null!;
    0 references
    public string emailID { get; set; } = null!;
    0 references
    public string homeAddress { get; set; } = null!;
    0 references
    public bool admin { get; set; }
    0 references
    public string password { get; set; } = null!;
}
```

Fig. 5: User Model

3. **UsersWaitList:**



```
public class UsersWishList
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]

    6 references
    public string? _id { get; set; }

    0 references
    public string userName { get; set; } = null!;

    [BsonRepresentation(BsonType.ObjectId)]
    0 references
    public string itemId { get; set; } = null!;
}
```

Fig. 6: UsersWishList Model

Routes and Controllers:

Items (Model - Item, Controller - ItemsController)

1. GET = https://localhost:7003/api/Items: Gets all the items/products from the database.
2. POST = https://localhost:7003/api/Items: A post method with an item request body (JSON) passed to add a new item/product to the database.
3. GET = https://localhost:7003/api/Items/{_id}: Gets the details of an item from the database.
4. PUT = https://localhost:7003/api/Items/{_id}: Updates the details of an item to the database.
5. DELETE = https://localhost:7003/api/Items/{_id}: Deletes an item from the database.
6. GET = https://localhost:7003/api/Items/waitlist/{_id}: Gets the waitlisted users of an item from the database.
7. PUT = https://localhost:7003/api/Items/waitlist/{_id}: Adds an user to the waitlist of an item to the database.
8. DELETE = https://localhost:7003/api/Items/waitlist/{_id}: Deletes an user from an item's waitlist in the database.

Users (Model - User, Controller - UsersController)

1. GET = https://localhost:7003/api/Users: Gets all the users from the database.
2. POST = https://localhost:7003/api/Users: A post method with a user request body (JSON) passed to add a new user to the database.
3. GET = https://localhost:7003/api/Users/{_id}: Gets the details of a user from the database.
4. PUT = https://localhost:7003/api/Users/{_id}: Updates the details of a user to the database.
5. DELETE = https://localhost:7003/api/Users/{_id}: Deletes a user from the database

UsersWishList (Model - UsersWishlist, Controller - UsersWishListController)

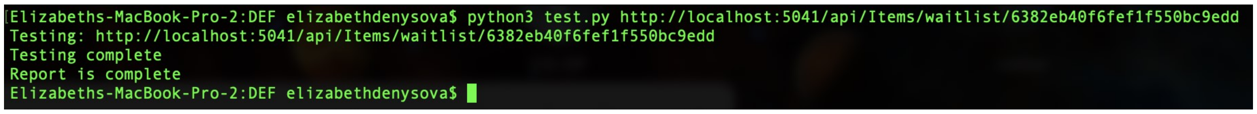
1. GET = https://localhost:7003/api/UsersWishList: Gets all the wish list items for all users from the database.
2. POST = https://localhost:7003/api/UsersWishList: A post method with a user request body (JSON) passed to add a new wish list user item to the database.
3. GET = https://localhost:7003/api/UsersWishList/{_id}: Gets the wish listed users of an item from the database.
4. PUT = https://localhost:7003/api/UsersWishList/{_id}: Updates the wish list of an item to the database.
5. DELETE = https://localhost:7003/api/UsersWishList/{_id}: Deletes an item wish listed from the database

Defense test:

Requirements: python/python3, python modules: sys, requests, HTTPError

Input: API get request to test for data exposure

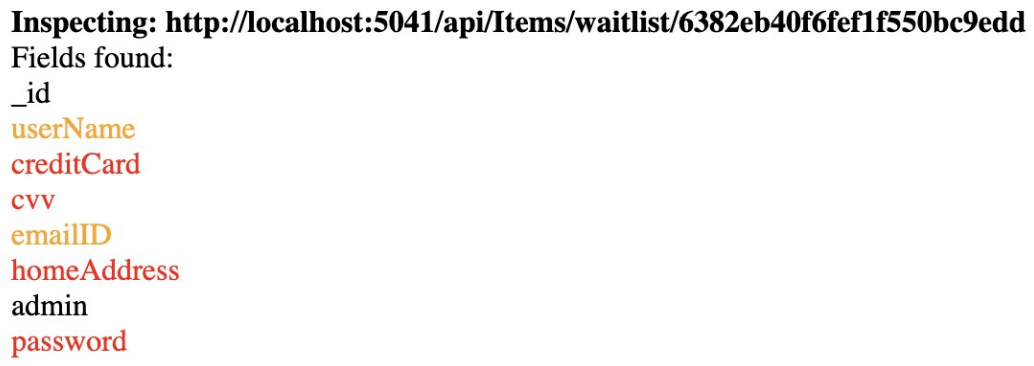
Sample execution:



```
Elizabeths-MacBook-Pro-2:DEF elizabethdenysova$ python3 test.py http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd
Testing: http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd
Testing complete
Report is complete
Elizabeths-MacBook-Pro-2:DEF elizabethdenysova$
```

Fig. 7: Python script

Sample report output:

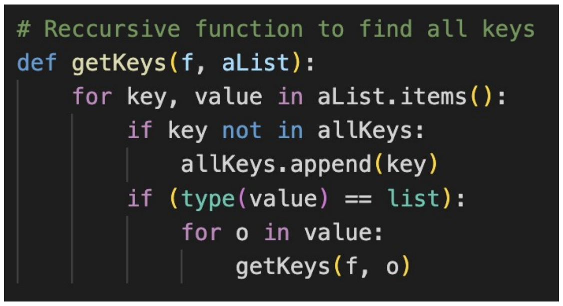


```
Inspecting: http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd
Fields found:
_id
userName
creditCard
cvv
emailID
homeAddress
admin
password
```

Fig. 8: Script output

More details in the defense demonstration section.

Purpose: receive URL of API GET request as command-line parameter, Get the data by calling the request, analyse data by using recursive functions to find all the fields (data) that is being returned. Color code the sensitive data and print the report

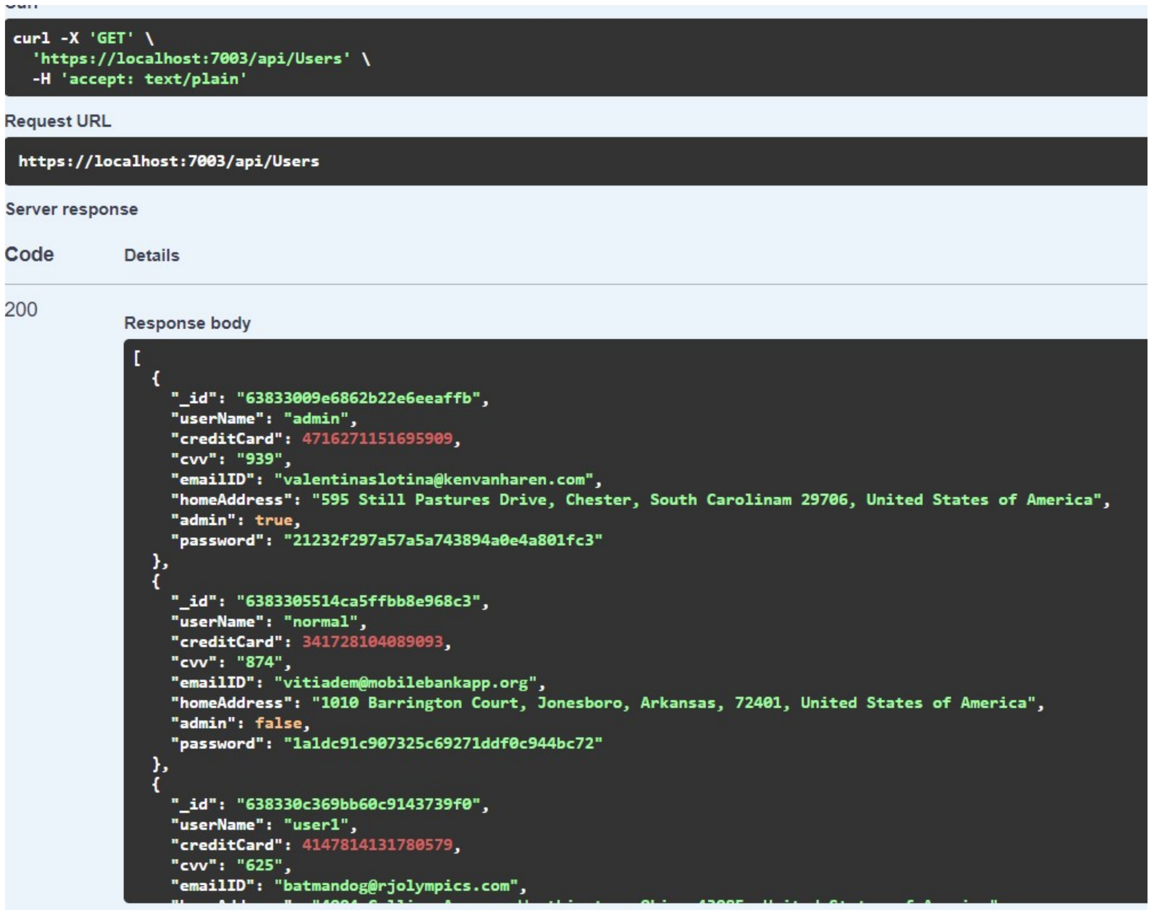


```python
# Reccursive function to find all keys
def getKeys(f, aList):
    for key, value in aList.items():
        if key not in allKeys:
            allKeys.append(key)
        if (type(value) == list):
            for o in value:
                getKeys(f, o)
```

Fig. 9: Code snippet

Attack Demonstration:

The online store keeps users' data in the database. Following image represents data that the User object consists of.



```
curl -X 'GET' \
  'https://localhost:7003/api/Users' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:7003/api/Users
```

Server response

Code Details

200 Response body

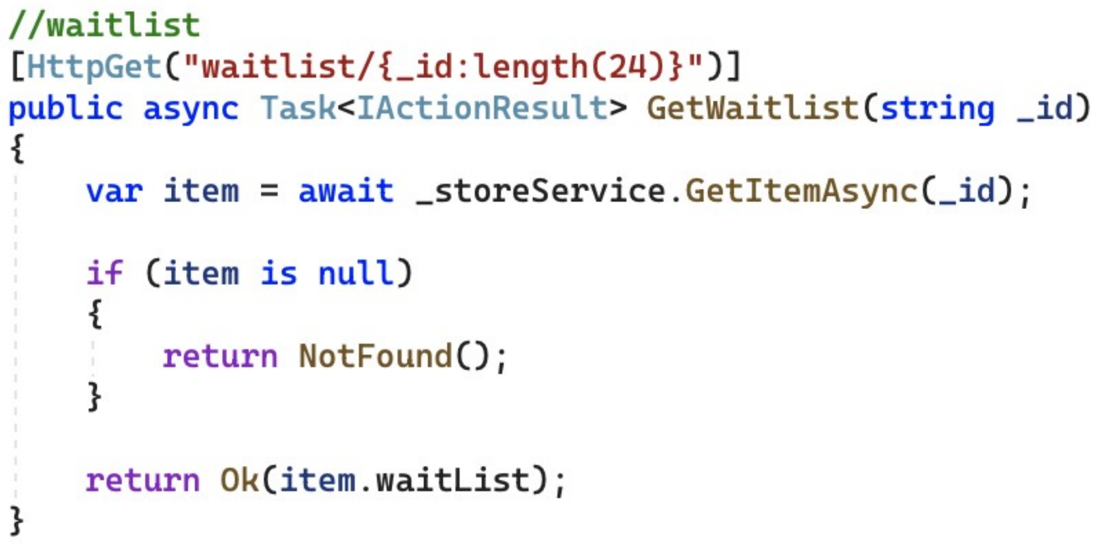
```
[
  {
    "_id": "63833009e6862b22e6eeaffb",
    "userName": "admin",
    "creditCard": 4716271151695909,
    "cvv": "939",
    "emailID": "valentinaslotina@kenvanharen.com",
    "homeAddress": "595 Still Pastures Drive, Chester, South Carolinam 29706, United States of America",
    "admin": true,
    "password": "21232f297a57a5a743894a0e4a801fc3"
  },
  {
    "_id": "6383305514ca5ffbb8e968c3",
    "userName": "normal",
    "creditCard": 341728104089093,
    "cvv": "874",
    "emailID": "vitiadem@mobilebankapp.org",
    "homeAddress": "1010 Barrington Court, Jonesboro, Arkansas, 72401, United States of America",
    "admin": false,
    "password": "1a1dc91c907325c69271ddf0c944bc72"
  },
  {
    "_id": "638330c369bb60c9143739f0",
    "userName": "user1",
    "creditCard": 4147814131780579,
    "cvv": "625",
    "emailID": "batmandog@rjolympics.com",
```

Fig.10: Excessive data exposure on users api call

This response contains a lot of personal sensitive information such as credit card information and address necessary to complete purchases.

Now consider the following request. The store has some items with limited availability so they offer customers to enroll themselves in the waitlist. The back-end was created to support this new feature and includes all current capabilities for the web application.

Each time users open the page with some item the request for that item is sent. The front-end verifies if the user is already on the waiting list or they might need to enroll.



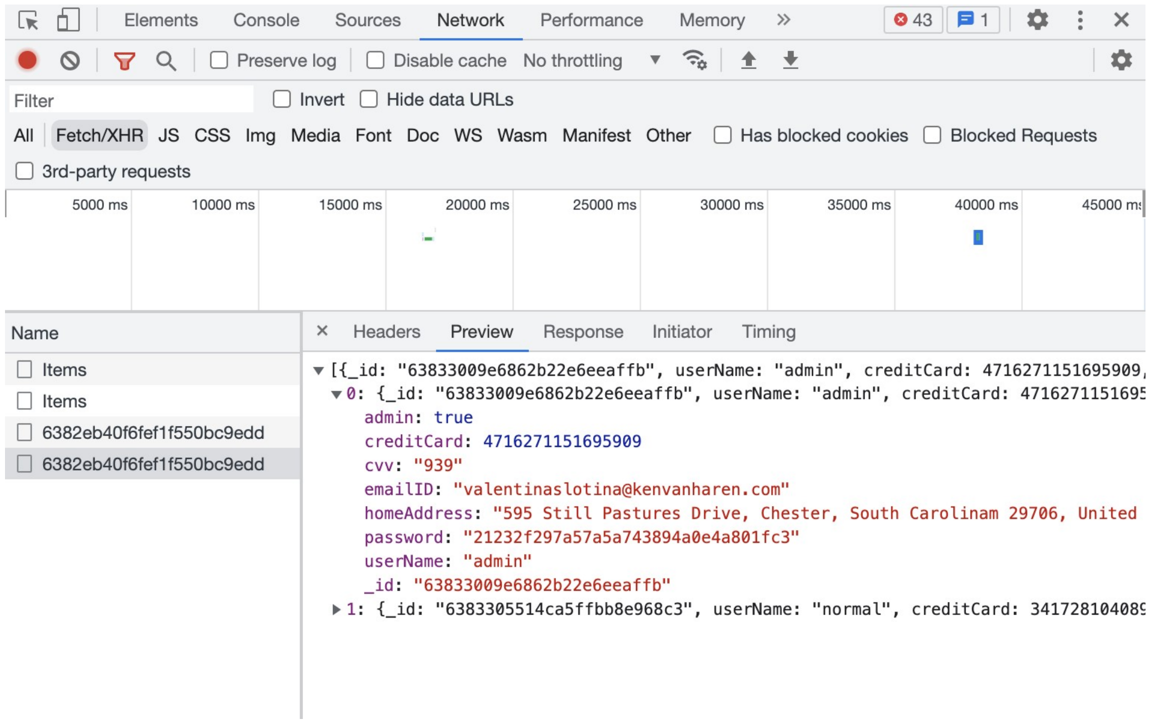
```
//waitlist
[HttpGet("waitlist/{_id:length(24)}")]
public async Task<IActionResult> GetWaitlist(string _id)
{
    var item = await _storeService.GetItemAsync(_id);

    if (item is null)
    {
        return NotFound();
    }

    return Ok(item.waitList);
}
```

Fig. 11: Waitlist api call

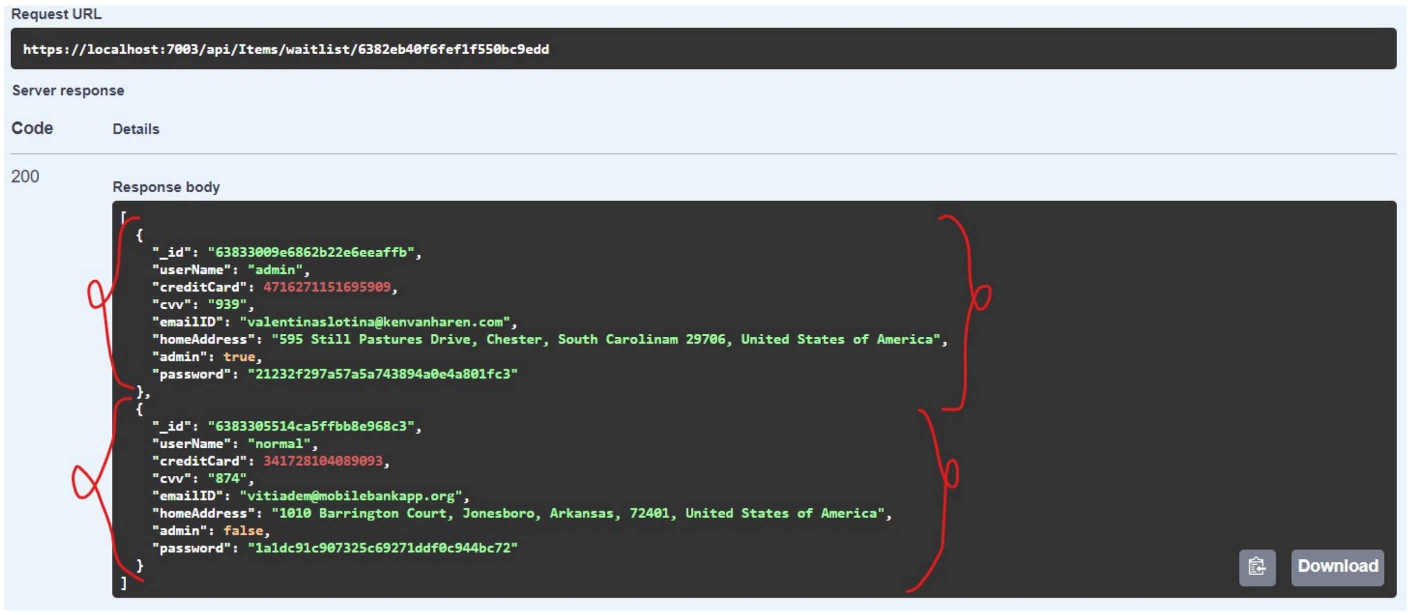
Now, if the user were to inspect the network and examine that *GET* request, they can get all the personal information of any customer in that waitlist.



```
Items
Items
6382eb40f6fef1f550bc9edd
6382eb40f6fef1f550bc9edd
```

```
▼ [{_id: "63833009e6862b22e6eeaffb", userName: "admin", creditCard: 4716271151695909,
  ▼ 0: {_id: "63833009e6862b22e6eeaffb", userName: "admin", creditCard: 471627115169
      admin: true
      creditCard: 4716271151695909
      cvv: "939"
      emailID: "valentinaslotina@kenvanharen.com"
      homeAddress: "595 Still Pastures Drive, Chester, South Carolinam 29706, United
      password: "21232f297a57a5a743894a0e4a801fc3"
      userName: "admin"
      _id: "63833009e6862b22e6eeaffb"
  ▶ 1: {_id: "6383305514ca5ffbb8e968c3", userName: "normal", creditCard: 34172810408
```

Fig. 12: API request inspection



Request URL

```
https://localhost:7003/api/Items/waitlist/6382eb40f6fef1f550bc9edd
```

Server response

Code Details

200 Response body

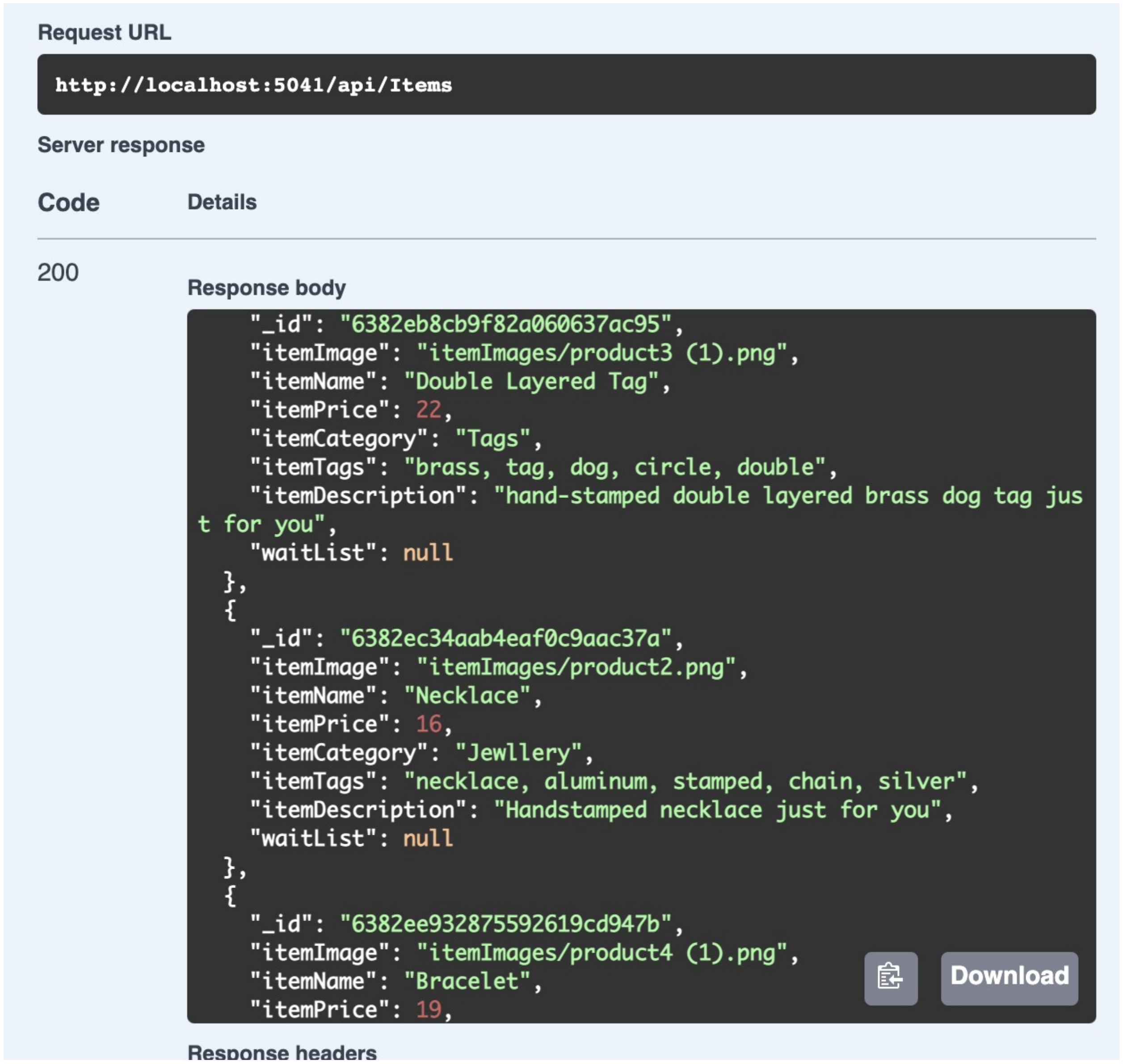
```
[
  {
    "_id": "63833009e6862b22e6eeaffb",
    "userName": "admin",
    "creditCard": 4716271151695909,
    "cvv": "939",
    "emailID": "valentinaslotina@kenvanharen.com",
    "homeAddress": "595 Still Pastures Drive, Chester, South Carolinam 29706, United States of America",
    "admin": true,
    "password": "21232f297a57a5a743894a0e4a801fc3"
  },
  {
    "_id": "6383305514ca5ffbb8e968c3",
    "userName": "normal",
    "creditCard": 341728104089093,
    "cvv": "874",
    "emailID": "vitiadem@mobilebankapp.org",
    "homeAddress": "1010 Barrington Court, Jonesboro, Arkansas, 72401, United States of America",
    "admin": false,
    "password": "1a1dc91c907325c69271ddf0c944bc72"
  }
]
```

Fig. 13: Data exposed in front end

Observation: When returning the item.waitList, it returns a list of User objects.

Defense demonstration:

What can be done to detect and prevent excessive data exposure? It is important to carefully examine data for each API call created but sometimes it is a very difficult and tedious task. Consider the GET request for items that the waitlist is part of.


Request URL

```
http://localhost:5041/api/Items
```

Server response

Code Details

200 Response body

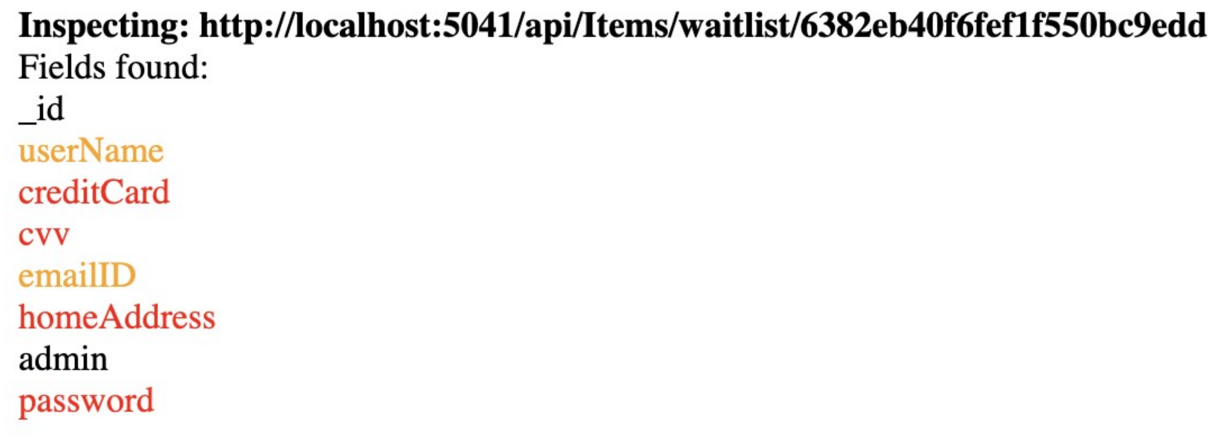
```
    "_id": "6382eb8cb9f82a060637ac95",
    "itemImage": "itemImages/product3 (1).png",
    "itemName": "Double Layered Tag",
    "itemPrice": 22,
    "itemCategory": "Tags",
    "itemTags": "brass, tag, dog, circle, double",
    "itemDescription": "hand-stamped double layered brass dog tag jus
t for you",
    "waitList": null
  },
  {
    "_id": "6382ec34aab4eaf0c9aac37a",
    "itemImage": "itemImages/product2.png",
    "itemName": "Necklace",
    "itemPrice": 16,
    "itemCategory": "Jewllery",
    "itemTags": "necklace, aluminum, stamped, chain, silver",
    "itemDescription": "Handstamped necklace just for you",
    "waitList": null
  },
  {
    "_id": "6382ee932875592619cd947b",
    "itemImage": "itemImages/product4 (1).png",
    "itemName": "Bracelet",
    "itemPrice": 19,
```

Fig. 14: Items api call exposes user personal information

There are too many fields and data to read through. In addition to this, there are empty waitlists for some items.
It would be better if this task was automated. Hence, a python script is developed to provide a report that helps analyze data being returned with any API request.
Navigate to the defense folder and run test.py with the command line parameter of API GET request url. For example:

python3 test.py http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd

It provides a html report (that can be converted to pdf) all unique fields that are in that GET request with color code corresponding to HIGH or MEDIUM data sensitivity. So for original version of waitlist GET request the report is:



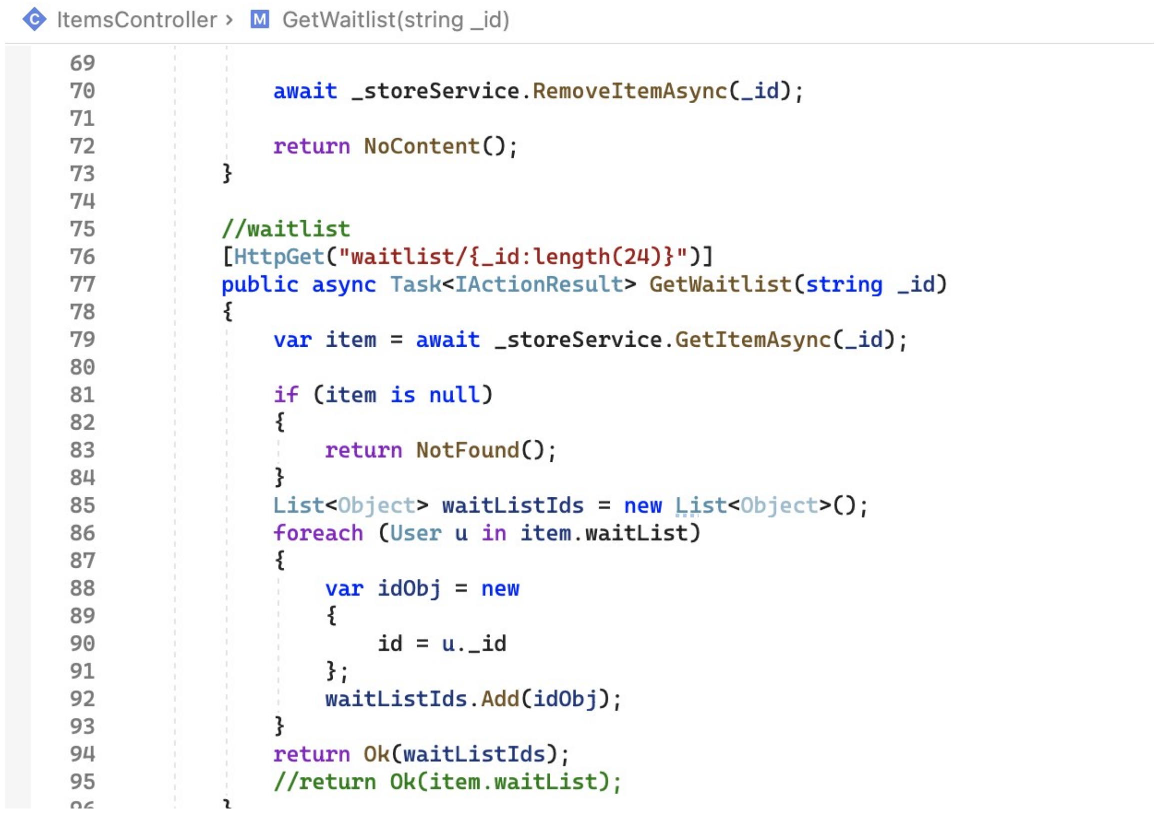
Inspecting: http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd

Fields found:

_id
userName
creditCard
cvv
emailID
homeAddress
admin
password

Fig. 15: Highly sensitive data exposed

Now it is easy to see that the waitlist object contains a lot of sensitive data.
Possible improvement is to return a list of users' ids instead of a full object. That way the front-end can still verify if the user is in the waitlist but no personal information is being revealed.



ItemsController › GetWaitlist(string _id)

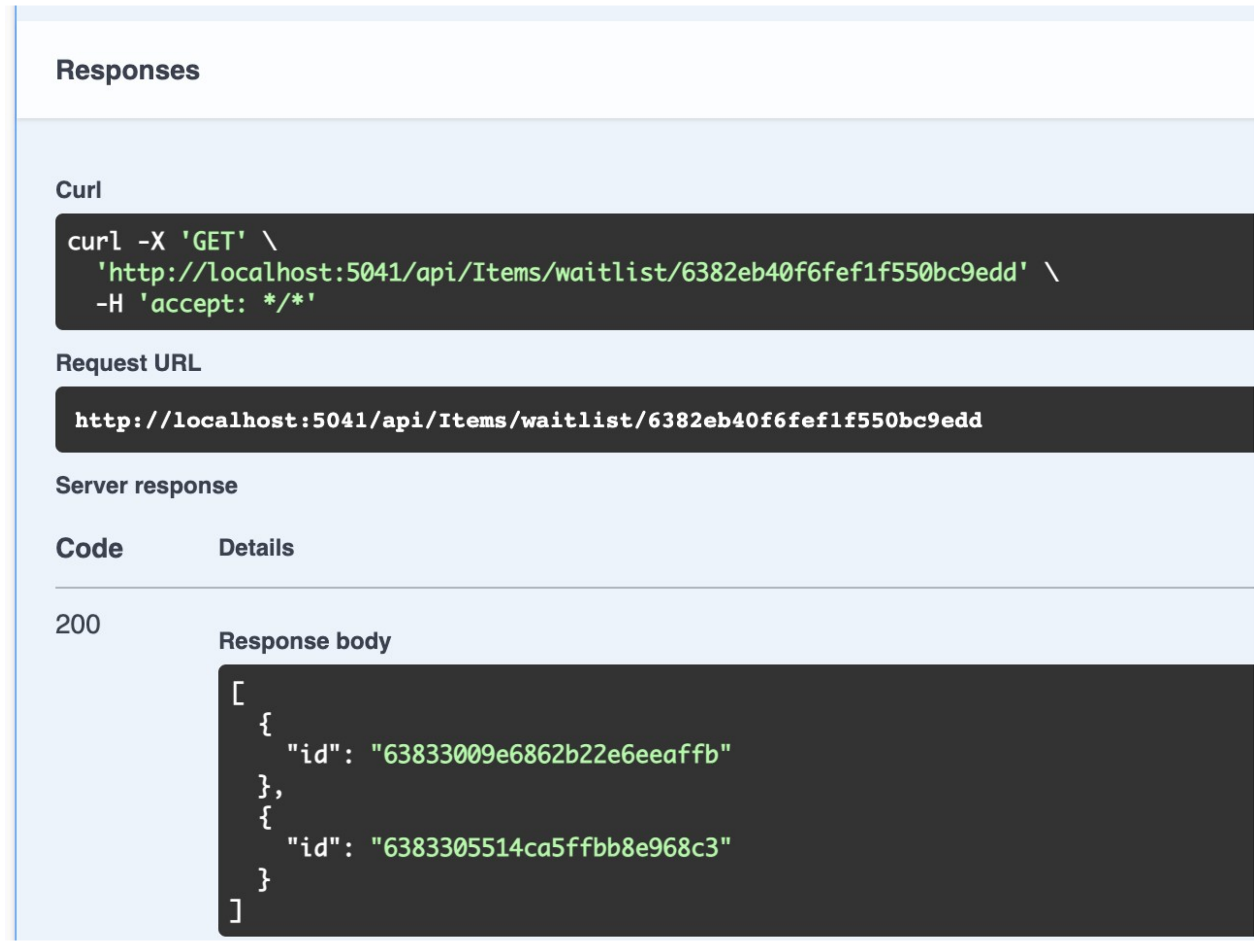
```
        await _storeService.RemoveItemAsync(_id);

        return NoContent();
    }

    //waitlist
    [HttpGet("waitlist/{_id:length(24)}")]
    public async Task<IActionResult> GetWaitlist(string _id)
    {
        var item = await _storeService.GetItemAsync(_id);

        if (item is null)
        {
            return NotFound();
        }
        List<Object> waitListIds = new List<Object>();
        foreach (User u in item.waitList)
        {
            var idObj = new
            {
                id = u._id
            };
            waitListIds.Add(idObj);
        }
        return Ok(waitListIds);
        //return Ok(item.waitList);
```

Fig. 16: Return user ids instead of personal information


Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd
```

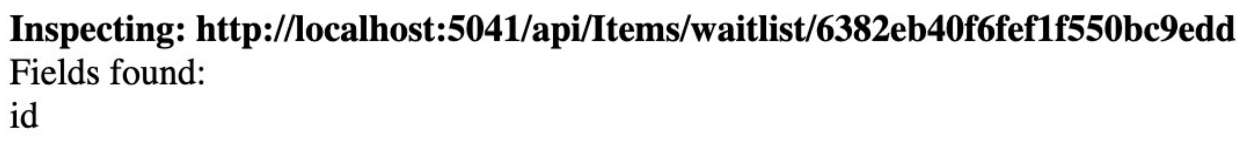
Server response

Code	Details
200	Response body

```
[
  {
    "id": "63833009e6862b22e6eeaffb"
  },
  {
    "id": "6383305514ca5ffbb8e968c3"
  }
]
```

Fig. 17: API respond to allow required data

Now, lets run the defense test again:
python3 test.py http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd



```
Inspecting: http://localhost:5041/api/Items/waitlist/6382eb40f6fef1f550bc9edd
Fields found:
id
```

This confirms that the modified API is safe and can be used in production.

Conclusion:

Typically, online stores and other websites store an overwhelming amount of data not only about their web application but also about their user base. This leads to a possibility of having a security flaw, more specifically, sensitive data exposure. Since many online stores keep sensitive information such as users' payment information, the exposure of such personal data to unauthorized parties can be devastating for the victims. Developers should be very careful with the use of data and API calls but there is always a chance of human error. In this report, it is demonstrated how an attack can take place when excessive data exposure vulnerability is exploited through a poorly designed API. A solution to this vulnerability was also proposed in the form of an automated test for API GET requests. This script can be easily modified to suit any specific needs of a web application and it can then be included in the testing flow before production release. This is done to limit human error in reviewing data and therefore further protecting customers’ sensitive information. Next, a defense test script was developed and its purpose was to find and fix the security flaw and to conduct a verification test to confirm that the issue was eliminated.

References:

1. Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/
2. Why is API security important? What is API security? (redhat.com)
3. Create a CRUD API using .NETCore. Tutorial: Building an ASP.NET Web API with ASP.NET Core | Toptal
4. MongoDB operations manual. MongoDB CRUD Operations — MongoDB Manual
5. CGI - Common Gateway Interface Script File Format. What is a CGI File?