**Take 1 — Registration and User-Authentication**



It is time to update the web application you created for *Dunedin Dairy* to include a facility to buy products.

The service back-end now consists of two separate services and associated RESTful endpoints. You have been using one of these services for the previous assignments. The second service, dealing with purchases, is new to this assignment. The help page for the second service is available at <http://redsox.uoa.auckland.ac.nz/dsa/Service.svc/help>.

The first service is open to all, and this is the service you used in the first version of the application. This service, in addition to the end-points you used earlier, allows a user to register at the application site. Don't we all want to track the users! **You will first add a new logical section to your app to implement this user-registration facility using the service endpoint that supports registration.**

The other service is related to the shop, and is only accessible to registered users. This service allows registered users to puchase products from the shop. Registered users are required to authenticate before using this service. For each of the shop items, add a 'buy now' facility in the shop . This can be crudely implemented by linking this 'buy now' to the corresponding shop service endpoint. The browser will then take care of authentication by prompting the user for credentials. Credentials of a registered user need to be supplied at this point. You could also use the pre-registered user jbon007 whose password is jbon007passwd.

User Authentication

Do you see why the simple approach of letting the browser take care of authentication is not ideal?

A better approach to authentication is to have a (logical) login section and direct the user to it when they want to purchase something (and if the user is not already logged in). The login section would collect the user's credentials (username and password) and use the credentials when making a purchase request to the server backend.

xhr.open("GET", uri, true, username, password);  
xhr.withCredentials = true;

Try implementing this logic, and see why the XHR request fails when used with credentials. We will find a fix for this later when we do the second part of this assignment.

If you have not done the first assignment, you can implement a shorter version that allows user registration, shop browse/search, and purchase. We do, however, strongly recommend that you do try implementing the complete app.

Please note the following requirements apply for the whole assignment.

1. You are *not* allowed to use any libraries or frameworks.
2. You cannot submit any images. However, you could hot-link images from other sources, if necessary.
3. The application should contain a single HTML page (named *UPI.html* where *UPI* is your UPI).
4. The application should work on any modern browser (such as Chrome, Safari and Edge).
5. The application should work on small screen devices.
6. There are submission requirements at the end of this document.

**Take 2 — User-Authentication and Pen-Testing**

In this part, you will use local versions of the services and data. To this end, please download copies of the open service, the closed service, and the data first. You also need to change the *Web.config* files in the two services to point to the local location where you have saved the data.

1. [Data](https://cws.auckland.ac.nz/CWS/CourseWorkService.svc/cwm?cid=DairyData). Unzip the data and place it in a folder such as *H:\335\DairyData*.
2. [Dairy Service (Open)](https://cws.auckland.ac.nz/CWS/CourseWorkService.svc/cwb?cid=DairySvc) . Unzip the service and place it in a folder such as *H:\335\DairySvc*.
3. [Dairy Service (Closed)](https://cws.auckland.ac.nz/CWS/CourseWorkService.svc/cwb?cid=DairySvcAuthenticated) . Unzip the service and place it in a folder such as *H:\335\DairySvcAuth*.
4. Now open the Web.config files in the two services, and find the line:  
   <add key="DataRoot" value="C:/WebRoot/Dairy/"/>  
   Change the value field to reflect where you saved the data. E.g., *H:\335\DairyData\*.
5. You can then run a service using the 64-bit edition of IIS Express. Example:  
   "C:\Program Files\IIS Express\IISExpress.exe" /port:8188 /path:H:\335\DairySvc  
     
   In Windows PowerShell, you need to prefix the above command line with & so that PowerShell can execute the command (rather than treating the command as a string):  
   & "C:\Program Files\IIS Express\IISExpress.exe" /port:8188 /path:H:\335\DairySvc  
     
   Test the open service by pointing your browser to <http://localhost:8188/DairyService.svc/help>. The browser is expected to show the API doc you saw at <http://redsox.uoa.auckland.ac.nz/ds/DairyService.svc/help>. Stop the service by quitting IIS Express.  
   The 32-bit edition of IIS Express – *C:\Program Files (x86)\IIS Express* – is not compatible with the supplied services. You should not therefore use it.

Now start the open service again using IIS Express. If the data and the open service are correctly set up, you should see a list of the products when you point your browser to http://localhost:8188/DairyService.svc/items. Note that the output shown in the browser may not be well-formatted, and in this case, you may wish to view the source to see the unformatted content.

You can test the closed service using the following command line:  
"C:\Program Files\IIS Express\IISExpress.exe" /port:8189 /path:H:\335\DairySvcAuth  
Test the service by pointing your browser to <http://localhost:8189/Service.svc/help>. Note that viewing the closed service requires authentication, and you could use the pre-registered user jbon007 whose password is jbon007passwd.

User Authentication

We noted in the first part of this assignment that a better approach to authentication is to have a (logical) login section and direct the user to it when they want to purchase something (and if the user is not already logged in). The login section would collect the user's credentials (username and password) and use the credentials when making a purchase request to the server backend.

xhr.open("GET", uri, true, username, password);  
xhr.withCredentials = true;

You would have also noted that the XHR request failed when supplied with user credentials.

XHR allows passing user name and password only if the origins match – i.e., the script that invokes the XHR and the URI the XHR tries to reach should have the same origin. This is called the *Same Origin Policy*.

Since you now have access to the shop service, you can publish your UI at the same location so that *Same Origin Policy* holds. Extend the UI you developed in the first part to incorporate this login logic (i.e., collect the user's credentials and use the credentials with XHR when making a puchase request) .

Q: Every time after we refresh the page, the user would be logged out automatically？

A: Yes - that's OK. If you want to retain the state, you can use <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

Here is how I did to avoid the duplicate binding issue:

* Put your web app folder (say CS335\_A4) inside the DairySvcAuthenticated folder
* edit your config. file:

             1 <system.webServer>

             2 <directoryBrowse enabled="true" />

             3 </system.webServer>

        add the statement 2 to your config file between <system.webServer>

* Start the server as per Assignment 4 Task 2 instruction
* Visit http://localhost:8189/ in your browser
* Go inside the CS335\_A4 directory and start your html page
* You should now see that your html page is starting with http://localhost:8189/CS335\_A4/your-UPI.html

Pen-Testing the Service

Threat Models

Threat models allow characterizing different levels of threats to a system. One of the commonly used model in network applications is the network/snooping/co-located user model.

**Network user.**

Network users are at the network edge. They are able to examine the contents of what they receive, and submit input that may not be legitimate. The invalid data they supply might not necessarily be intentional. Possible attacks include SQL injection, XSS, and CSRF.

**Snooping user.**

Snooping users have access to the network. They can not only examine the contents of what they send and receive, but also the contents of what others send and receive on the network. In addition, they may be able to modify, drop, or duplicate the contents of what others send and receive on the network. Possible attacks include session-hijacking, denial of service, and replay.  
A classical snooping user requires physical access to the wired network. With Wi-Fi networks, there is a greater risk of snooping users. Open Wi-Fi networks allow all traffic to be snooped upon. WPA2-PSK (Wireless protected access — pre-shared key) encrypts the traffic, but anyone who has the PSK can decrypt.

**Co-located user.**

Co-located user is someone or something with access to the user's machine (or server). E.g., a malware on the user's machine. Co-located users have the ability to measure or interfere with IO (e.g., key strokes, displays, drivers, etc.), and read and/or modify the filesystem and/or memory. The level of intrusion depends on the access level (e.g., admin level grants all the privileges). Possible attacks in this category include Trojan horses, key loggers, and data theft.

It is thought the service has a number of vulnerabilities. You are now asked to penetration-test the service, and report your findings. You will first assume the role of a network user who is able connect to the service with the view of injecting code. You will then assume the role of a co-located user with access to the data to give more insight into further potential vulnerabilities. 要求您对服务进行渗透测试，并报告您的发现。首先，您将扮演一个网络用户的角色，该用户能够通过注入代码连接到服务。 然后，您将扮演一个位于同一地点的用户的角色，可以访问数据，以进一步了解其他潜在的漏洞。

Note that if you corrupt the data during your pen-testing, you can reinstate a fresh copy from what you originally downloaded.

The backend services cache the output data for performance reasons. If you corrupt the data, you may also need to re-start the services to clear the server cache. Similarly, a browser may cache data, again for performance reasons. So you may also need to clear the browser cache. Browser caches typically are persistent. Consequently, re-starting the browser is not likely to clear its cache. You need to look at the browser settings to find out how to clear its cache.

Please answer the questions below, first assuming the role of a network user, and then assuming the role of a co-located user. Where the questions require you to write an HTTP request, please provide a URL if the request is GET; if the request is POST, please provide the complete request including all the necessary HTTP headers. E.g.,

POST http://www.site.org/foo.cgi?t=Find HTTP/1.1  
Host: www.site.org  
Content-Type: text/plain  
Content-Length: 5  
  
Hello

Network User

Assume now the role of a network user, and answer the following questions.

**N01**

Find out where SQL injection is possible within the service APIs. Write down an HTTP request that injects SQL.

**N02**

Find out where HTML injection is possible within the service APIs. Write down an HTTP request that injects HTML.

**N03**

Using HTML injection above, write down an HTTP request that inserts an arbitrary image into the UI you constructed.

**N04**

Write down an HTTP request that leads to executing a user-supplied script.

**N05**

The services do not stop cross-site request forgery (CSRF) attacks. Describe in a single sentence how a cross-site request forgery attack could be mounted on the service.

**N06**

Combine the CSRF vulnerability you found above in N05 and the XSS vulnerability you mentioned in N04 in an HTTP request. Write this request down.

Co-Located User

Assume now the role of a co-located user, and answer the following questions. This will require you to study the data used by the services. Note that if you corrupt the data during your pen-testing, you can reinstate a fresh copy from what you originally downloaded.

**C01**

Open the database file using a binary editor (e.g., [frhed](http://frhed.sourceforge.net/en/) on Windows or [0xED](http://www.suavetech.com/0xed/) on MacOS). Use the [magic number](https://en.wikipedia.org/wiki/Magic_number_%28programming%29) in the file to find out the name of the database management system. Write down the name of this database management system.

**C02**

Get a tool that is able to open the contents of the above database, and examine the contents of the database using this tool. How many tables exist in the database? (Note that you can download both Windows and MacOS versions of the tool.)

**C03**

One of the tables in the database contains user credentials. There are two users in the supplied database, one is jbon007, what is the name of the other?

**C04**

Find out the password of the user you mentioned in *C03* above. The password is thought to be a common dictionary word. For example, you can find a list of commonly used words [here](http://www.wordfrequency.info/free.asp).

WWW-Authenticate: Digest realm="Dairy", nonce="42fb4d88e0ddfdc290fd936db007ef00", opaque="5878731121398187"

**C05**

In the data pack the service uses, there is a folder containing large images, and the service does not publish any of the images from this folder. Craft an HTTP request that can still give a network user access to one of the images in this folder. Write down this HTTP request.

**Submission**

Please submit to the [Assignment Dropbox.](https://adb.auckland.ac.nz/) the following items.

1. The following source files of the application. (Note that anything else is not permitted.)

a. The HTML file (called *UPI.html* where *UPI* is your UPI).

b. The CSS file (called *UPI.css* where *UPI* is your UPI).

c. The JavaScript file (called *UPI.js* where *UPI* is your UPI).

d. The SVG logo file (called *UPI.svg* where *UPI* is your UPI).

【可以交A2的那个svg logo file】

1. A *plain text* file (called *UPI.txt* where *UPI* is your UPI) containing your answers to the pen-testing questions.