DC Assignment 2 Report

Bharath Sukesh

19982634

May 23rd, 2021

1. Design Choices

2.1 Three-tier database application – Tutorials 1 – 3:

This application consists of a .NET Remoting Server, a WPF Client application, as well as a Data Tier that represents the database, I made in Tutorial 1. Tutorial 2 comprised of adding a Business Tier to this solution, and 3 we make a Web Server to act as the Business Tier. The Business Tier .NET Remoting Server formed in Tutorial 2 was as simple as making a new Remoting Server with a port different to the coexisting Data Tier server, and then making a connection via an interface to the Data Tier's methods. This means that the business tier had very little *additional* functionality to the Data Tier, other than the feature to search for a client. The Data Tier's database was formulated via random strings – I had two string arrays consisting of vowels and consonants, and I would retrieve in a for loop, a random index from those two string arrays to form random data. Similar randomized approach was used with the other fields for each user.

My Business Tier was fairly default for this program; its search functionality was asynchronous as asked by the worksheet; however, I applied an index out of range fault contract, which is thrown and caught whenever I try to get a value out of range. This is handled by my `IndexOutOfRangeFault`. The Web API variant of the program had a very similar business tier with the same exception handling, and remote interface calls, just over a Web API using controllers instead. I attempt to catch null references, and other format exceptions, for e.g., format mismatches when trying to insert a string instead of an integer for the index search, or trying to enter a number for the last name search – this validation was performed via Regex. If any error were to occur, it would produce output in both the console in Visual Studio, as well as the Console for the .NET Remoting Servers via Logging, which was nothing extravagant – simply implemented as suggested in the Worksheets.

1.2 Three-tier Bank application – Tutorials 4-5:

This application consists of two Web API's: one for Data Tier, another for Business Tier. The Business Tier is tied in with the Display Tier, as our Business Tier's functionality is mapped out onto a web page which the worksheet asks us to manually design. I handled user exceptions in the Business Tier, meaning the any invalid input can be caught, and should not stall or crash the program. This once again involved steps like checking whether the parameter is the correct type the method needs via either Regex, or TryParse() or in other scenarios, checking whether the provided ID is valid, and so on. One problem I had to overcome was due to the nature of the DLL, Process Transactions will remove any submitted transactions from a common/shared List that holds them (embedded in the Bank DB DLL). This meant that if I had called ProcessTransactions() every time I formed a *valid* transaction between two users, then it would process it, alter the balances of each user, and remove it from the List (DLL nature), thus I whenever I would use that Transaction ID in GetTransactions() to view it's details, the transaction would be null as it no longer resides in the List. To overcome this, I made it so that it will only process transactions, every 3 transactions. This does mean that the likelihood of a transaction actually being processed with no problems could be a little less (not the function running, rather the likelihood of money being transferred due to amount balances possibly not being enough after 1-2 transactions), but it meant that I can get the Transaction ID from a Visual Studio's Console output, and it would return a valid set of details regarding that transaction, before it gets processed.

As for my presentation tier, I decided that to avoid any issues with lack of accounts, when I make a user, I also make an account. This means that no user will be left without a corresponding account. I felt this would easier for the User, and would not cause any further issues for that user's experience. Everything else was fairly default for the presentation tier in Practical 4, a page for each system functionality. Practical 5 asked us to perform load our pages onto one home/index page. In order to make my page look nice, I decided to add a background, and an image (referenced from Freepik), and provide some colour to the page so it looks appealing. Other than the aforementioned error handling performed in the Business Tier, I also performed error handling in this project at the Website level via Ajax queries. When one is sent from the website to my Biz Tier Controller, depending on whether the operation was successful (return from those methods), I catch an error at JavaScript level in my `.cshtml` files; checked via simple if statements. I then universally across all website functions provide an output to the Console Log (Inspect Element → Console) as well as Alert dialogs in some cases to inform the user upon the completion of a request indicating its success. Extensive logging/WriteLine's to the Debug Console means that Users can know what is happening, and when. These exceptions being handled in the Business Tier means that rarely do they even get to the Data Tier, thus avoiding any potential harm *should it happen*.

1.3 PEER-TO-PEER BLOCKCHAIN APPLICATION – TUTORIALS 6-8:

The peer to peer application allows us to have a board of jobs circulate among Clients, with each client having a differing port number allowing only the clients that didn't submit the job, to be able to complete a job. I performed validation on the code that can be submitted such that it must contain an appropriate python method header containing main(): as well as proper indentation and ensuring it returns something. The code (job) also only be submitted if the textbox's contents is neither null nor empty. Upon submission, I provide a status to the client that submits the job, indicating its progress.

As for the job computation itself, I catch exceptions such as:

- NullReference Exception, for if the script is null
- UnboundLocal & UnboundName Exceptions, for if the code is invalid
- Runtime & SyntaxError Exceptions, for there are any runtime or syntactical errors held.

These exceptions are indicated to the user via a Message Box.

When a client leaves, normally it would throw an EndpointNotFound exception, so to deal with it, I catch this exception and call a method to remove them from the P2P chain. I had struggled a little with synchronisation between the Job Server, which I used Mutexes when adding a Job, to resolve.

In Practicals 7 & 8, I wanted to try using Delegates as we did earlier in the semester for the blockchain application. Most of these practicals were attentively following the Worksheets; the server thread and the miner thread coexist and work in unison. To avoid mismatch of clients, I again caught an exception in a while loop to ensure that two clients are on different ports at the very least. I used a queue that is statically manipulated by the miner delegate, enclosed in a while(true) loop. I felt this was the easiest way of going about it. The Transaction Generator would call a method in my Miner class, to add a Transaction object created by the client, into the static Queue. This addTransaction process was enclosed in a mutex at Client side to ensure no synchronisation issues can occur where to can add at the same time. The worksheet asks us to implement cryptographic hashing and base 64 encoding, which seemed to be straightforward and did not tank the performance of my program. I decided to use static objects a bit more in Practicals 7 & 8 as it seemed a *lot* easier to keep track of. A lot of client-side exception handling prevents users to add abnormal transactions.

---

2. XSS

2.2 Persistent XSS

Triggering:

Persistent XSS consists of when the attacker injects a piece of malicious code/script into the website directly and thus the website database, meaning that all victim users that connect to that website server/database will thereby download the website's contents including the malicious script. The source of malicious code is stored server side at the website. For example, this can be emulated by having a user connect to a website that has been infiltrated. This would trigger the script written by the attacker, which for example would steal the victim's cookies as a parameter and send to the attacker's server, to which he can extract, and manipulate however they like.

Preventing:

As this type of attack is stored at the Database side, input validation is required server-side to ensure bad things do not enter the system. An example would be to firstly check if the request is empty or null, followed by checking if there are any special characters present. If so, deny and throw an exception, if not, allow the request to proceed. This way any malicious requests cannot be submitted as malicious JavaScript can't be executed.

2.3 Reflective XSS

Triggering:

Reflective XSS consists of when the attacker attempts to trick a victim into using a maliciously modified request URL for a given website, in order to steal information. It is triggered by the user using this request that contains the attacker's script, which would be processed as a valid request i.e., the victim would send the request for a given website, and the website would treat it as a valid GET request from the victim, and *reflect* it back to the victim's browser. It then runs in the victim's browser, which runs the script written by the attacker, thereby giving the attacker access to the victim's information.

The source of the malicious code is embedded in the initial GET request from the victim to the website, which is unobserved by the Website server; it gets ignored/overlooked and treats it like any valid GET request, and then bounced back to the client, which will run the website's response script (creates a HTML with the response). *Now* the malicious script gets run. The website for example, would only check whether some property of the get request is non-empty/not null, e.g., a search query; once it realizes it is non-empty, it will reflect. Another thing to note is that the victim usually won't be able to realize that they are sending a request containing a malicious script; usually this is hidden (which is why the server accepts it as a normal valid request). The attacker can either send the link (containing a malicious script embedded in a request) or advertise it on his own website.

Preventing:

This can be done by checking input fields upon submission, and once again performing input validation at server-side, to check for special characters that are typically in JavaScript. If it does find these special characters, I would Encode them such that the JavaScript would be ignored, meaning it cannot execute JavaScript code at client-side.

2.4 DOM based XSS

Triggering:

It starts off similarly to a reflected XSS; attacker advertises a link containing the malicious script to the victim(s), which then would trick the victim into initiating that request. Upon this request being made, like Reflective, it goes to the website and runs its valid code. However, unlike reflective where the website has the malicious script, here it doesn't create a new HTML, rather it would provide the victim (and hence the attacker) the actual code that resides server-side (response script), which is a valid response. The problem lies in that it loads the JavaScript victim-side, and upon loading, can transform the JavaScript from the server's HTML *into* the malicious HTML designed by the victim and thereby executing the script designed by the attacker. It is not executed when a page is loaded as part of the HTML from the Server; rather executed sometime after the valid page loads.

Preventing:

By having the JavaScript check its inputs, which I would do via Regex to check for special characters. If there are any present, I would halt execution of that input field, meaning no attack can be run. This means that nothing can be injected into the page that should not be there. For example, given an attacker tries to send a malicious string to the Server, the server would never receive this string as input handling is done at the client-side (JavaScript).