CSCI 4271 Report

Freddy Melo, David Smith, Jaime Somero

System Design

Badly Coded Bookmark Manager (BCBM) is a cross-browser and cross-device bookmark management program. This system utilizes a local database for the client, an HTTP application server (BCBMC), and a cloud server (BCBMS). The user is only able to access the client HTTP application server, interacting with the cloud server through the webpage hosted by the HTTP server. This webpage enables them to add, delete, and order their bookmarks into a top 10 list. The BCBMC client interacts with third-party websites to acquire details such as the website's name for the bookmark's name. These bookmarks are stored in their local database. With the introduction of BCBMS, clients can synchronize between their local database and the cloud server's database. To access these features, the user must create an account by registering an email address and password for authentication. The user then receives an email with a link to visit, which activates their account with the password they registered with initially. The data flow diagram below shows the flows between the user's local client and the BCBMC cloud server.

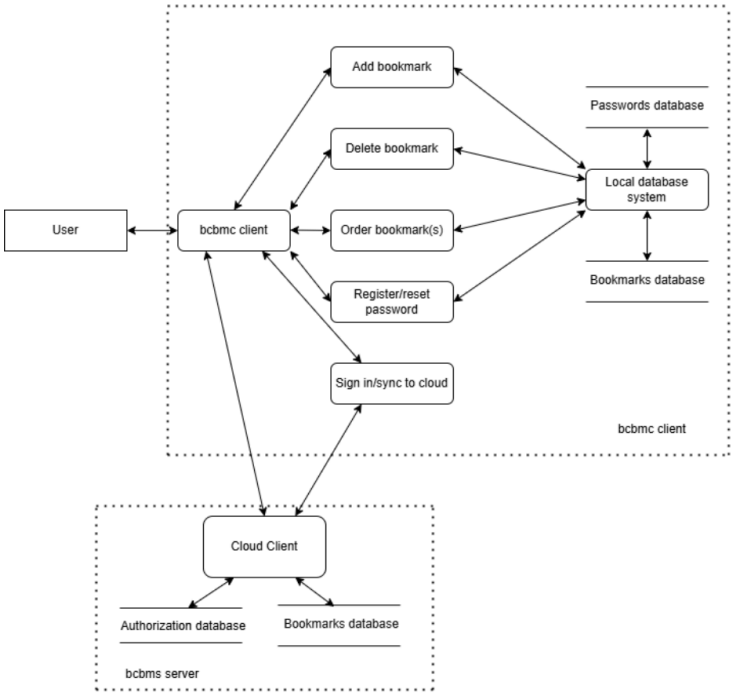

Figure 1: Data Flow Diagram (DFD) for the Badly Coded Bookmark Manager program

Threat Model

What does/should it do?

The web browser client application should allow the authorized user to add, delete, and order bookmarks in their local database system. The user will be prompted to add a rank for it, title, id, and url. The sign in/sync to cloud process should allow them to sync their bookmarks with a cloud database. They can do so after they have registered as a user, and will use a username/password to log in, except for when their computer has a token from a previous log in. These processes should be protected against any potential common threats including spoofing attacks, tampering with code, and elevation of privilege. To do so, each point of interactivity from the client should be carefully analyzed to ensure no vulnerabilities go undetected.

What could go wrong?

Individuals with malicious intent could be looking for methods to surpass given application privileges and cause internal harm. This can be completed through various avenues including spoofing user credentials, injecting harmful code to override given permissions, and any other methods that exploit the program's vulnerabilities. Within the program, a few vulnerabilities jumped out such as the authentication code session handling, overflow attacks, URL injection threats, and encryption handling. Furthermore, once an exploit is known it can easily snowball into other exploits being uncovered. If users know their data and privacy is not well secured within the BCBMC program as a result of these exploits, it could lead to the user base avoiding it. All vulnerabilities must be corrected to prevent attackers from corrupting the program.

What are we doing about it?

To minimize the threats and vulnerabilities to the program, the following analysis report was created to evaluate the security of the program. The steps involved consisted of inspecting the given program's code and evaluating how each file interacts to create the functioning website. Then, six vulnerabilities within the code were identified, and checked for exploitation through multiple tests to create proof of concept programs for each. These proof of concept programs explain how the vulnerabilities can be exploited through a series of instructions. These programs can be found within the project's Github repository: https://github.umn.edu/melo0035/CSCI-4271W-Project. The purpose of the analysis is to make the BCBMC system more secure and reduce potential attacks that could be carried out against the program. This report summarizes the findings and offers some possible mitigation strategies to counteract the vulnerabilities.

How did we do?

The team did a well-rounded job offering improvements to be made for the program's security. This first step of the process involved locating the six most crucial vulnerabilities. Presented in the findings below, the first vulnerability details how the authentication code session protection can be easily brute forced by an attacker. Due to the lack of IP binding or session management outside of this code, if an attacker finds a valid code, they can access that user's session. This is proven through the given proof of concept (ID 1) exploit attack that demonstrates the legitimacy of the vulnerability concerns. A possible mitigation was provided to correct the vulnerability within the program. The second vulnerability (ID 2) examines how the lack of safe measurements for the user input such as rank could lead to overflow

attacks, a proof of concept for its defense, and mitigation. The third vulnerability (ID 3) covers how to execute code remotely using a malicious URL input of a webpage controlled by an attacker. The fourth identified vulnerability (ID 4) examines how the encryption process of the file stored in the cloud is faulty and can easily be decrypted using brute force to expose a major information disclosure vulnerability within the sync functionality. The fifth vulnerability (ID 5) looked into how the generation of tokens to authorize an account's email were easily able to be spoofed, leading to fraudulent accounts being made with victim's emails. Lastly, the sixth vulnerability (ID 6) focuses on the possibility of command line injection if an attacker is able to get the user's browser to send a valid GET request to the BCBMC server. Thus, the team effectively researched and provided six improvements that could benefit the program.

S.T.R.I.D.E. In Threat Model

Spoofing: The Sign in/sync to cloud process could be the victim to a classic user spoofing threat. A hacker could try impersonating another user through methods such as password guessing, password cracking tools, or stealing it through physical means. This could lead the hacker to impersonate a user to sign in and access private user information. Furthermore, the authentication code was found to be easily spoofed through bruteforce as the lack of security for the code and the low amount of possible combinations make it a possible attack. Spoofing could also be possible through fraudulent token authorization if an attacker is able to generate a token for a victim's email without them knowing it.

Tampering: A possible network tampering target could be the data flow between the account authentication process and the cloud client process. If the network that the user is utilizing the bookmark manager from is unencrypted, an attacker could possibly modify data over the network while it is being transferred to the cloud client, especially since this flow crosses between two trust boundaries. There might be tampering vulnerabilities in the sync function with the cloud database, if an attacker sends overly large bookmark names to the cloud they could possibly overwrite other users' bookmarks or the same might apply using overly large usernames in the create account function. Another possible tampering attack would be a SQL injection that would either edit or delete the databases containing user information, which would be devastating to the user base of the website. There is also a possibility for spoofing, where using a victim's email the attacker could generate the authentication token and then authenticate an account that doesn't belong to them.

Repudiation: A possible source of repudiation threats could be the fact that there is no log to track user's actions within the program. Attackers could add or delete bookmarks on the system and then claim that they never actually did due to the lack of a log recording which functions were used while they were signed in to their account.

Information Disclosure: Possible sources of information disclosure threats could be the sign in/sync to cloud process as well as the register/reset password process. Both of these processes utilize an email-password combination in order to authorize a user's account, and these can often lead to information disclosure if there are improper error statements. We need to make sure that when an incorrect email or password is input, the system does not display which field was incorrect as that is a common information disclosure problem. Information disclosure vulnerabilities could also possibly be exploited if an attacker is able to access the html file that contains the cloud information for all users' bookmarks. There is also an attempt to encrypt the bookmarks that are stored within this file, however there could possibly be flaws in the encryption and if an attacker is able to find the key to decrypt this information, they could view other users' saved bookmarks. Information disclosure threats could also be made possible through SQL injections that could utilize the SELECT feature to grab usernames and

passwords from the server databases which would be a major security threat if the authentication method being utilized is not secure.

Denial of service: A possible source of a disk denial of service threat could be the bookmarks database in the local database trust boundary. This is because if the disk is filled up by an attacker repeatedly adding bookmarks, it would completely fill memory and lead to errors, crashes, or memory corruption on the user's local computer. Possibilities for denial of service are also possible by an attacker repeatedly uploading large http files to the cloud database by using the sync function. This could crash the network, and if the attacker keeps sending the requests could possibly prevent the server restart function from working. Deletion of the user databases as mentioned in the tampering section could also be classified as a denial of service, as it would prevent users from being able to access their accounts on the website.

Elevation of Privilege: The add bookmark process could be victim to elevation of privilege threat through the override process. If a hacker were to find a specific link to surpass the string specifications, they could inject their own malicious code. This would be an elevation of privilege as they would be able to modify the code beyond what their given permissions.

Summary of Findings

Vulnerability ID 1: Weak authentication code

When a user initializes a BCBM session, they are assigned a randomly generated 6-digit authentication code. The process begins in lines 70-79 of **bcbmc.cpp**, where a code is generated and made to be the user's session identifier. The session path becomes: "https:localhost:8888/given_code/". Lines 89-140 in **bcbmc.cpp** utilizes this path for every user action such as adding, deleting, or viewing all bookmarks. The current iteration of BCBMC does not link the authentication code to a specific user's session or have any security measures to protect the user outside of this line (**bcbmc.cpp**): `if (c != code || code == -1) res.code = 401;`. The authentication code is treated as a global variable, meaning anyone with the code is technically the user. Therefore, if an attacker correctly guesses this authentication code, they will gain access to a user's session. The amount of authentication codes that can be generated is very low based on these lines:

```
read(urand, &randseed, 1);
// Make it extra random with hash function
randseed = (randseed * 0xa3d7 + 0xd3ad) % 0xFFFFF;
```

The read line only reads 1 byte, this means the code is restricted to 2^8, which evaluates to 256 combinations. In a breakdown of the second line, `randseed = (randseed * 0xa3d7 + 0xd3ad) % 0xFFFFF;`. Converting the values from hexadecimal to decimal, 0xa3d7 becomes 41,943, 0xd3ad converts to 54,189, and 0xFFFFF turns into 1,048,575. This means the codes are calculated through the equation: ((value 0 to 255) * 41,943 + 54,189) % 1,048,575). `while(randseed > 999999 || randseed < 100000)`, further narrows down the codes to meet the 6-digit restriction and ultimately results in 21 authentication codes. These 21 codes are calculated through a short Python script, **generate_auth_codes.py**, which is included in the Github repo for evidence. The revised PoC, **new_auth_code_brute_force.py** runs through the significantly shorter

list of the 21 possible combinations until it hits a valid code. Once a valid authentication code is found, the attacker can use it for easy access to a victim's session.

Mitigation:

A good mitigation for this vulnerability would be to bind the given authentication code to the client's IP address. When a client begins their BCBM session, their authentication code should be connected to their IP address by the server. Within, **bcmc.cpp** `clientIp = req.remote_ip_address` should be implemented. The following line, `if (c != code || code = -1) res.code = 401;` should then be changed to, `if (c != code || code = -1 || req.remote_ip_address != clientIp) res.code = 401;`. Therefore, anytime a new request is made for a new page, the server will always verify the request is coming from the original client IP address. Even if an attacker guesses the code, they still would not be able to get into the session as their IP address does not match the authentication code's binded IP.

Vulnerability ID 2: Rank Overflow

The website's top 10 bookmarks page was created to show a user's favorite bookmarks ranked from 1 to 10. However, any proper check of these rank values was not implemented. The process of this vulnerability begins in **manage.c**, starting at line 36 `write_favorites()` and ending at line 160. These lines detail how the website processes the rankings. The website assumes the user will simply input a rank 1-10, however there is no safeguard in place for when an attacker decides to input a higher or negative value. Therefore, if an attacker were to give a rank of anything greater than 10 or less than 1, for example: 99. It would cause an out-of-bounds array writing to occur. Although the function, void insert, in manage.c only loops for the rank values 1 through 10, the line: `(arr[rank] = newval;)`, always occurs. This is why any value above 10 will still go through to cause a heap overflow and memory corruption. Refreshing the website would not fix this either because this is written into permanent files of the running user session.

Thus, within the github repository under vulnerability ID 2 is a PoC, **overflow.html**, that illustrates this effective attack. A malicious html page is given, the attacker must input the victim's authentication code in the script. Then, the attacker runs the html script, which contains the malicious overflow add bookmark. This PoC simulates how an attacker could send a victim running BCBMC this harmful link and once they open it, it will automatically add a bookmark to google.com with a rank of 99 to the user's top 10 bookmarks. Now, when the user navigates back to their top 10 list their page should look corrupted.

Mitigation:

This attack can be mitigated through putting a line such as: `(if rank < 1 || rank > 10)` before the loop occurs in **manage.c**, line between line 106 and 107. This one line or an implementation of this logic should safeguard against an overflow. Since it shouldn't run the loop if the input value the user entered is greater than 10 or less than 1. Therefore, any variation of an implementation that checks the value being within the expected ranks of 1-10 will mitigate this attack.

Vulnerability ID 3: Malicious URL

In **manage.c**, the `prepare_add_form()` function interacts directly with external HTML web pages via curl to find the title of the page for the bookmark. It does not check or escape the contents of the title in any way before entering it into the HTML of the add.html form page. This means that an attacker who is aware of the vulnerability and has control of a webpage that is bookmarked can directly affect the

contents of the **add.html** page. If the title begins with "**'>**" (a single quote and a greater-than sign), the <input> tag and value attribute are closed, and any following HTML in the title will be executed by the browser as if it were from bcbmc, and not from an external source.

In our proof of concept example, we were able to use this vulnerability to tamper with the add form page and execute javascript code in the user's browser without appropriate permissions. The title of the page is:

```
evilpage'><script>window.location.assign("https://www-users.cse.umn.edu/~somer137/good.html")</script>
```

This redirects from the intended form confirmation page to an entirely attacker-controlled page, which would display within the BCBM page and could run any additional javascript without memory or other constraints (such as with the title). An attacker could additionally spoof the expected BCBM page and convince a user to input and send user information or otherwise approve a dangerous request (thinking that it's coming from a trusted site).

Mitigation:

One way to mitigate this issue would be to encode the title input in the C code of `prepare_add_form()`. Specifically, it would be necessary to encode/escape characters such as **<**, **>**, **'**, and **"** into HTML character entities to display, rather than as characters that could/should be executed as code. We were able to find more specific information on the HTML character entity codes on the W3schools site here: https://www.w3schools.com/html/html_entities.asp. This would, for example, turn each "**<**" into "**<**", which would be displayed as a "**<**" in the original form input, rather than executed as code.

Vulnerability ID 4: Encryption

In **sync.c**, there is a vulnerability in terms of how the bookmarks are encrypted when they are stored in the cloud_all.ehtml file. The main cause of the vulnerability comes from line 205 `(generate_symkey` function) of **sync.c**, where a value is read from the urandom file and is then used in the linear congruential generator to create part of the symkey. This process is done twice, to create two parts of a key utilizing random values from the urandom file and then combining them into one number which will be used as the key that can encrypt and decrypt the **cloud_all.ehtml** file. The vulnerability comes from the fact that only 1 byte is read in from the urandom file each time, and for each byte that is read, there are only a possible 256 values due to 8 bits in a byte only having a possible number in binary of 0-255. Since this is done twice, that means that there are only a possible 256 * 256 = 65,536 possibilities for different keys, since each number is multiplied by a constant value in the code and also added to another constant value, which doesn't affect the possible number of solutions. 65,536 different keys is not very many and can be brute forced using a python script very easily to find the key that will decrypt all users' bookmarks.

For the proof of concept document there are two files, the encrypt_message.c file and the **DecryptBookmarks.py** script. The c file is run just for the proof of concept to create an example **cloud_all.ehtml** file simply containing the text "<html>", since that text will be in every html file. The file is then encrypted using the exact code from the sync.c file in order to ensure that it follows the same process and is encrypted in the exact same fashion as if it were done by the bcbmc program. The

cloud_all.ehtml file that is created is simply an example file that is encrypted exactly how it would be in the program, ensuring that if my python script were used on the real cloud_all.ehtml file that is pulled from the cloud in the bcbmc sync operation, it would work the same way. Now that there exists a file that is encrypted the same way as the sync.c function would, the python script **DecryptBookmarks.py** can be run and it will decrypt the file and place the decrypted text into the try.html file. The script simply iterates through all 65,536 possible keys that can be created and attempts to decrypt the **cloud_all.ehtml** file with each one. If after the decryption process is completed with the current key, the file contains "<html", then we know that the file has properly been decrypted and can print out the key and exit the loop.

Mitigation:

A possible mitigation for this vulnerability in the encryption process would be to read more than just 1 byte from the urandom file when generating the key, which would significantly increase the key entropy and make it much harder to brute force a key. Line 205 in the **sync.c** file of bcbmc-sync-main directory could be edited so that the read function would be: `read(urand, &r, sizeof(r));`. The 1 that is currently in the amount to be read argument of the read function could be replaced with `sizeof(r)` which would instead read enough bytes to cover all possible values for a long integer, which is 2^32. This amount of possible values is significantly larger than the original 1 byte read, which only had a possible 2^8 values. This increased entropy of values means that the possible number of keys would go from 65,536 to over 1.8 x 10^19 possible keys, which is currently an infeasible number to solve for with brute force as it would take far too long.

Vulnerability ID 5: Token Generation and False Authentication

Tokens can be easily created and found using brute force in order to create false accounts for a victim's email and verify it without being able to login to their email account. This is made possible due to the fact that in **regdb.cpp**, only 1 byte is read into the `t_lo` and `t_hi` variables from uran (lines 130-131). This is identical to the problem with vulnerability ID 4, where even though 8 bytes should be read in, it is only 1, which causes the same exact problem of low token entropy that can be easily brute forced since there are on 65536 possible options. (See vulnerability ID 4 for the calculation.) This low number means that tokens can easily be generated using brute force and tested against an existing account that has not been email authenticated yet in order to find the corresponding token and register the email. This is a major spoofing vulnerability and a very big security threat, as an attacker could complete the registration process using a victim's email, and then run the **TokenGeneration.py** script in order to authenticate the email and create the account, even without having access to the victim's emails. This means an attacker can create false accounts for any email address and pretend to be other user's and use their email accounts without them even knowing it.

The python script that I wrote to exploit this vulnerability first makes a request to the registration page in order to begin the registration process for a victim, in this case "example@email.com". Once the account is successfully registered and the authorization email has been sent, the script proceeds to loop through all 65,536 possible tokens using the same generation technique that was used in the **regdb** file. Every 100 tokens, it will print an update saying how much progress has been made. Once the proper token has been found, it will exit the loop and print out the token for the victim's account as well as the seeds that were used to generate it. After the program has finished executing, there will be an existing account that has been authorized with a token using the victim's email address and they won't even realise it.

I calculated the average number of tokens able to be checked per second by this script and it was around 7, which means that in the worst case scenario, it could take around 1 hour and 18 minutes. While this may seem like a long time to a regular user, this is a relatively short period of time for an attacker to conduct a brute force attack, and is easily able to be completed.

Mitigation:

A possible mitigation for this vulnerability would be the same as for vulnerability 4 again, where they could change the 1 in lines 130-131 of **regdb.cpp** to an 8 (the size of the `t_lo` and `t_hi` variables) in order to increase the range of tokens that can be generated to a number that would take far too long to be a realistic vulnerability. The increased range of possible tokens would make the brute force process go from 78 minutes in the worst case, to an amount of time that would be impossible to realistically justify the attack with.

Vulnerability ID 6: Command injection

If an attacker is able to get a user's browser to send a valid GET request to the BCBMC server, they can send a request which imitates the request made when the user adds a bookmark to their BCBMC database. The `prepare_add_form()` function in **manage.c** is called when the server receives this kind of request, and is vulnerable to command injection. This is because the `URL` variable in `prepare_add_form()` is not checked, escaped, or validated before it is added to two separate curl commands and sent to the system to be carried out. This means that an attacker is able to control the content of those commands and is able to escape the commands to inject their own command to carry out along with or instead of the intended curl commands.

Before the attacker is able to do this, the attacker needs the authentication code generated by BCBMC for the user's current session. As detailed in previous vulnerabilities, this is not particularly difficult to do. The attacker can either test for a valid authentication code before sending the request, or can have the user attempt to send a request with every possible authentication code, using a brute-force method to carry out the attack.

In the proof of concept example, the attacker attempts to disrupt the user's ability to use the BCBMC client by removing all HTML files from the BCBMC's main directory. They do this by having the user's browser send the following request:

```
http://localhost:8888/<authcode>/add/url/%27%3B%20rm%20%2A.html%3B%27
```

Then, provided `<authcode>` is replaced by the valid authentication code, the BCBMC client decodes the "URL", which becomes `'; rm *.html; '`. This is injected into the curl command, instructing the user's system to remove all **.html** files from the working directory, and closing the single quotes from the original curl command.

Mitigation:

This attack would, of course, be much harder if authentication codes were harder to find and/or validate, but the focus of this vulnerability is the command line injection. One way to mitigate this attack would be to check the validity of the user input URL after decoding it and before adding it to the curl command. There are a number of ways to do this, but one way would be to use regular expressions to make sure that the user input follows the standard URL formatting, or to use a different method to send a request to the given URL input without sending the request through the command line. This would also have the added benefit of making sure the user is only adding bookmarks that direct them to real sites.

Provided again below is a link to the Github repository containing our proof of concept programs and README.md document detailing how to utilize them: (https://github.umn.edu/melo0035/CSCI-4271W-Project).

Addendum: Group Accountability (Project 1)

The review will detail the contribution of each group member in order from the top to the bottom of the report. For each member's focus in terms of what we drafted on the report, we decided that David would focus on the design section, Freddy would focus on the threat model, and Jaime would focus on the vulnerabilities. Even though we had sections that we were responsible for when it came to writing the report document, we collaborated on almost all sections and ensured we all committed equal time and effort throughout the project as outlined in the following addendum. The system design portion was written by David Smith and Freddy Melo. David drafted the original paragraph and Freddy revised it as time went on. The data flow diagram was collaborated and created equally between Freddy and David. They both worked simultaneously alongside each other to ensure the completion, with David adding a few revisions after the creation. The threat model and its corresponding sections were drafted and refined by David and Freddy once again. The specifics are available through the Google Docs history tracker, but it once again was a balanced collaboration between the two.

For the first vulnerability, Freddy identified the potential of the file vulnerability and began the initial research into it. He drafted a rough version of a proof of concept exploit, possible mitigation, and the layout that would present these findings alongside the detailing paragraph. David refined the vulnerability details, ensured the exploitation was legitimate, and corrected the mitigation. Freddy used David's corrections to create the bash script and finalize the vulnerability. The process was an equal collaboration between the two.

For the second vulnerability, Jaime Somero identified the second vulnerability and did the corresponding research into the issue. She formed the proof of concept for the exploitation and necessary mitigation. Furthermore, she wrote all of her findings into the allocated vulnerability section and refined it. All three group members reviewed the details and made any necessary changes to fit cohesively with the report.

The addendum was initially drafted by Freddy Melo, then reviewed and finalized by David and Jaime. In terms of the project process, Freddy contacted David and Jaime after they responded to a Piazza post in search of group members. David created and shared the group Google Doc, in the first meeting. Jaime set up regular weekly Zoom meetings for the group to collaborate together and set duties. Freddy Melo created and shared the Github repository for the proof of concept codes. Then, every task was split amongst whoever felt like they had a stronger understanding of the task. If additional help was needed, group members asked each other for help, leading to the collaboration throughout the entirety of the project. This resulted in a respectful, productive, and responsible environment that helped accomplish the project by the deadline.

Revisions and Group Accountability (Project 2)

For this second iteration of the report, we decided that Freddy would focus on the design section of the report, Jaime would focus on the threat model section, and David would focus on the vulnerability section. Just like in the last submission, while we each focused on our section in terms of revisions and additions, we all worked together as well to brainstorm and come up with solutions and revisions to the whole document.

Freddy added to the design section of our report so that it was more fitting to the new code that was released with the second iteration, and David revised the DFD which was the only part of the design section that required revisions from project 1. The DFD was changed so that the local database stores were within the same trust boundary as the local client, the trust boundary was renamed to bcbmc client which is a more accurate descriptor than what we originally had, and a data flow was added between the local and cloud clients, which were the only suggestions made for the original DFD.

For the threat model section, the information within the “what could go wrong” section was made more concise. The purpose for the “what are we doing about it” section was updated to be a more accurate description of the second report, those were the only two edits required within the threat model section from the first revision. The team also added descriptions for the two new vulnerabilities we uncovered within the “how did we do” section, as well as changed the descriptions of the vulnerabilities ID 1 and ID 2. The first two vulnerabilities had to be changed since our original vulnerabilities on project iteration 1 were declared to be out of scope. Furthermore, we added more possible threats to the STRIDE per element analysis to include possible vulnerabilities that could be within the scope of the new code released for this iteration.

For the vulnerabilities section, we first had to change both of our original two vulnerabilities from the first iteration of the report since they were marked off for being out of scope. Freddy, with the help of the team, found two new vulnerabilities to replace the original two as a revision to be within the scope of the project. Jaime then found new vulnerability number 3 and David found new vulnerability number 4, which focuses on the new part of the released code. Each of us created our respective proof of concept programs within the github repository and added our own descriptions of how to run each program in order to showcase our findings.

Revisions and Group Accountability (Project 3)

For this iteration of the project, found in this document, cs4271-project3-smi02247-melo0035-somer137 , we agreed to have Jaime focusing on the design section, David on the threat model, and Freddy on the vulnerabilities section. As before, all sections were somewhat collaborative. This means the team did divide the work into sections, however, help and assistance from other team members was provided wherever else was necessary.

In the design section, Jaime, David, and Freddy all contributed to the updated DFD, adding in content from the third iteration such as the third-party websites inclusion and an internet browser. Other adjustments were made to the general structure based on grader feedback from iteration 2. Additional information was added to the details to better correlate with the revised data flow structure. There were also some general revisions and tweaks made to the design section, as well as to the overall report, in terms of writing and structure, largely done by Freddy. These were the only edits and revisions that were deemed necessary for the design section.

The threat model section was largely revised by Freddy, who formalized the language and added new information for clarity and depth. Most revision was done in the “What could go wrong,” “What are we doing about it,” and “How did we do” sections. David added some possible vulnerabilities to the “STRIDE in Threat Model” section, to improve the depth of analysis. Jaime and David also edited the “How did we do” section to reflect the new vulnerabilities added to the report.

All group members were involved in revising and adding to the vulnerabilities section in different ways. Freddy revised vulnerabilities 1 and 2 according to grader feedback. This included narrowing down the brute force attack for vulnerability 1 and offering an alternative mitigation. Vulnerability 2 was altered to be delivered to the user opposed to the user running a weird input. David found vulnerability 5 and detailed it within the summary of findings section. David was also responsible for writing the proof of concept script for the vulnerability and adding it to the GitHub repository. Jaime found and added vulnerability 6 to the summary of findings. She also wrote the proof of concept code for vulnerability 6.

Freddy and Jaime made some adjustments to the overall report, especially in the vulnerabilities section, in order to make the style more consistent across the entire document. This addendum was originally written and drafted by Jaime, with feedback and edits from the rest of the group.