Team Name: Celebration of Capitalism
Semigroup: 921/2
Assigned Part: “All things evil”

I. Subscription Service – Data acquiring, validation and storage:

Design and implementation of a page/tab/form within the application, to be used by the users of the social media platform to subscribe to our premium services.

This implies a frontend in which the user can insert the credit card number, owner’s full name, expiration date and CVV.

A backend is also required for validation (e.g. checking that the card does not expire on 13/2032 (non-existent) or 09/2011 (already expired), or if the card belongs to an existing processor, such as Visa or Mastercard), for storing the credit card information to the database and for marking the user who filled the form as a premium user (e.g. a separate table with premium user IDs only).

II. Subscription Service – Post priority, Block override, Group override:

Design and implementation of the subscription service’s features.

Post priority – A premium user’s post could have a different weight associated with it if the post system uses a priority queue, so those will always come out first; otherwise, a system akin to Instagram’s or Reddit’s sponsored posts (where every few posts an ad appears) could be implemented using a queue, which will be displayed before the one that handles posts from non-premium users.

Block override – A premium user can send private messages/DMs to any non-premium user, even if they were blocked by said user. If a user’s ID belongs to the premium user ID table, a call to the messaging API will be performed if and only if the non-premium user has the premium one blocked.

Group override – A premium user can search for any open or private group, and join it, even if it’s invite only. If a user’s ID belongs to the premium user ID table, a call to the group API (specifically the part that will handle joining a group) will be performed. In case the group is hidden and invite-only, visibility options could be tweaked, using the aforementioned group API.

III. Relation analysis and discord

This system should be capable of analysing the relationship between users based on a quantifiable "interaction metric" – which is a number representing the significant time spent by user X interacting with a certain user Y.

At its core, an in-memory graph is used to analyse this relationship, built from existing data regarding a user's friend list and their direct messages to other users.

We define $r: (U, U) \rightarrow R$ a function with domain pairs of users mapped to some real number.

Provided we have two graphs, we will remark that for two users u1, u2:

$$r(u1, u2) = 0, u1, u2 \in U, u1 \neq u2$$

implies there is no edge connecting u1 and u2 in any the graphs.

We define with UF(u) and UD(u) the set of users which a user u is friends with, or which they have messaged, respectively.

Define two more functions $uw: U \rightarrow N$ defined as the user's weight in the algorithm. The function has the following value for a user u from U:

$$uw(u \in U) = card\big(UF(u)\big) + card(UD(u))$$

i.e. we define the weight function as the total number of friends plus the total sum conversations a user has — thus more active users will have a higher weighting in our system.

Define the function $m: (U, U) \rightarrow N$ the function which retrieves the number of messages sent between two users.

Our system may also have 3 optional "adjusting factors", which are just 3 real constants that the developers or system administrators may change to adjust the importance of each parameter in the relationship function, denoted by f1, f2, and f3 respectively.

Thus, the following is true:

$$r(u1, u2) = f1 * uw(u1) + f2 * uw(u2) + f3 * m(u1, u2)$$

The resulting graph is generated as such:

1. For all users we are interested in analysing (it could be only "premium" users, for example), create a vertex in the graph, and generate all pairs of the relationship function.

$$r(u1, u2), \forall u1, u2 \in U$$

2. Make an edge between all users where the relationship function did not result in 0.

3. The weight of the edge between two users is equal to the value of the relationship function between those two users.

With this groundwork foundation laid down, we can start to perform relation analysis between users. Some examples of what you can do:

1. For a user X, look at all the neighbours in the graph (i.e. all other users which have an edge with our user X) and find the edge with the minimum weight. This would signify the "least desired relationship", and the user X could be prompted to send a message to this least desired user in order to "not burn bridges", resulting in higher activity on our platform.

2. For a user X, look at all the neighbours in the graph. Find the neighbour with the highest user weight (the uw function) and then prompt user X to send a message to not "ruin their relationship with the popular guy".

3. For a user X, look at all the neighbours in the graph. Find the edge with the highest edge weight (i.e. the highest relationship function value) — a relationship to user Y, let's say.

   Then, do the same for user Y. If it happens that user X and user Y do not share the same edge, then it means that user Y is paying more attention to some other user not X, hence user X would be prompted to bother user Y further so as to not "get ignored".

IV. Malicious Subscriptions – Company contact, service acquiring and storage

Design and implementation of an unlisted (can be accessed only by using a link) page/tab/form within the application, to be sent to companies that expressed their interest to use our platform to push their services/newsletters/subscriptions.

This implies the creation of a frontend in which the company can specify the type of service they want to push and any other information related to the chosen service (e.g. their APIs to the service, so that our platform will perform operations on it in point V's functionality).

A backend is also required for storing the company and its service to the database. (Unlike at point I, where validation was required so that we get a valid credit card, it should fall upon the company that wants to do business with us to send valid information, so validation restrictions can be more relaxed here).

V. Malicious Subscriptions – Non-consensual user subscription to sponsored services

Design and implementation of the non-consensual subscription system and the service severity system.

This implies choosing a small sample of users from the database (around 1-10%, depending on the "severity" (e.g. a paid subscription is more severe than a newsletter push) of the chosen service) to subscribe to one specific company's services.

These operations should be achieved by using the APIs provided to us by point IV's functionality (since this will be sandboxed, simple mock APIs will be designed; for example, for a newsletter service, we'd just need a function that adds the email to their database; we've done our part as the middleman between the company and the user, and the call to push emails to their subscribers (and our users) should be performed by the company alone)

Another detail to take into account is the "severity" of the service pushed by the company. Newsletters can be considered harmless enough to be pushed onto a lot more users, and big overlaps between the chosen user samples should be avoided, so we can cover a larger portion of the user base. Paid subscriptions are very harmful to the user, thus the sample size should be a lot smaller compared to the newsletter one, and big overlaps between user samples is recommended, to at least maintain the façade of an isolated issue.

For example: between 3 isolated users, a situation where User 1 has 3 paid subscriptions non-consensually assigned to him and User 2 and User 3 have 0 is preferable to a situation where all 3 users have a paid subscription each (this sample size is very small for this, a more accurate example would be 1 affected / 100 total). By

keeping this isolated, the majority of the user base is left unharmed and unaware of the more “severe” services we’ll push onto them.

i.e. All users have an equal chance to be chosen for services of specific severity, say we choose 1000 users for the newsletters and 100 for paid subscriptions. These users will have their IDs stored in a new table, signalling which users were chosen to have non-consensual subscriptions pushed onto them. The connection is then established and we subscribe these users to their services.

The next time we have to fulfil a newsletter subscription quota (and we take 10% of our user base) the table that stores user IDs for users already subscribed has data inside. Say we must subscribe another 2000 users. We could take 20% of the entries of the table (or the entire table if 20% of said amount exceeds the total number of entries) to subscribe to this service, and the rest of 80% will be chosen randomly between the users that aren’t in the table.

Similarly, the next time we have to fulfil a paid subscription quota (and we take 1% of our user base), we’ll already have users that are subscribed to other paid services. Say we must subscribe another 200 users. We could take 90% of the entries (but by our previous example, 180 exceeds 100, so only 50% will be fulfilled), and the remainder will be chosen randomly between users that aren’t in the table.

VI. Scam Bot Service

Design and implementation of a system that will create scam bots in order to steal data from legitimate users.

This comprises two parts: designing a mechanism for the interaction between the bots and the users, as well as an algorithm to generate said bots into our social media platform.

Fake accounts can be created by generating fake, but convincing user profiles and registering them in the platform's database, such that no verification will be required. We will also have a database dedicated to fake accounts, in order to prevent these accounts from messaging each other, as that would be redundant.

Once the accounts have been created, we will run a program that will select a number of legitimate users (for example, 1%), and these users will receive a message containing malicious links from the scam bots.

The behaviour of our data thieves can be customized in multiple ways:

- we can specify the number/percentage of users that will be targeted;
- we can select an intermission time for the "attack" waves: we don't want our bots to overwhelm the platform, so in order to prevent this, we will schedule the assaults (for example, the bots will send messages every 6 or 12 hours)
- we can designate a maximum number of messages per wave for each bot: if one user receives messages from multiple bots, they will be less susceptible to the scam; we want to prevent these collisions by establishing a balance between the number of messages and the number of selected users
- they will be able to select template messages from a database, which will allow us to insert new scamming methods and to remove ones that have been used too often, or ones that are too old

In the case that our platform has a user reporting feature, we will run a periodical check every 12 or 24 hours that will omit banned accounts from the algorithm and create new ones. In order to stop the program from creating too many accounts, we will establish a number threshold that can be either arbitrarily set, or that can scale with the legitimate population of the platform.

VII. Scam Bot Service

We can make full use of the fake accounts by creating the web pages users will visit by ourselves. Users will be asked to insert their account details, as well as credit card information in order to receive their advertised "prize". Once we obtain all this

data, we can store it in our database and make use of what we have acquired in the *Malicious Subscriptions* feature.

This part's focus is on frontend, and as such, we will have to design websites that are convincing. Success can be achieved by creating faithful clones of popular sites, such as "x.com", with just one addition: you have to login using our platform's credentials. Other than that, the sites will not implement other fields with which the user can interact.

We can also create websites that will only ask for credit card information, and store it in a database that can be used by our other divisions (to make purchases on the Black Market or Deep Web, for example).

For the backend, we will use what we have created in our previous services, as there is nothing new that is needed by this feature.