HumboldtX

System Design

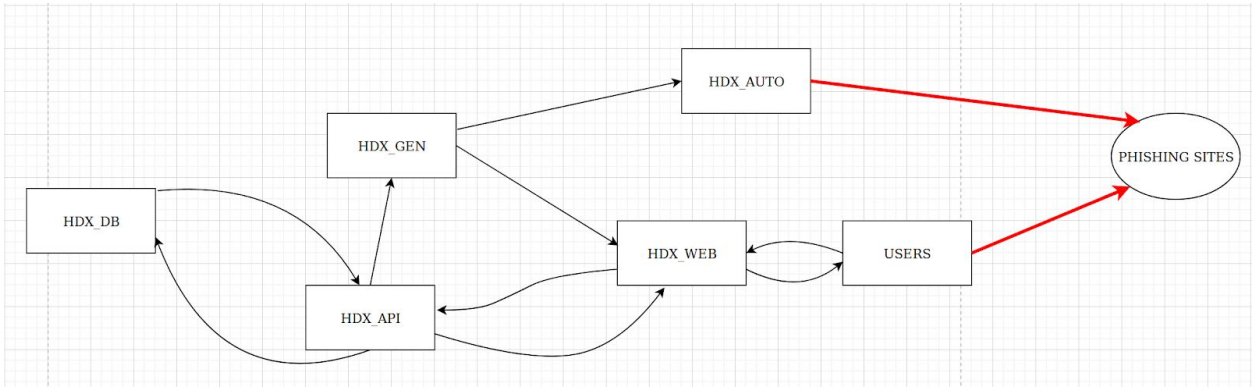
Ricardo R. Williams II

***Abstract*- This paper will briefly introduce the system design of one possible implementation of the Humboldt 2.0 ecosystem. First the overall goal will be discussed in case the reader is not familiar with the Humboldt project. Then the system design will be covered, followed by an explanation of the database schema.**

I. Introduction

The Humboldt Project outlines a system that can be leveraged to combat phishing campaigns via an aggressive counter measure. Most anti-phishing methods rely on a reactive approach. Humboldt on the other hand will utilize people to submit fake credentials to phishing sites. In an effort to halt ambitious phishing campaigns against an institution by flooding the phisher's data with indistinguishable fake credentials. For more information on the background on which this paper is being written, please refer to the papers "Crowd-sourced Phishing Disruption with Humboldt" and "Leveraging the Crowds to Disrupt Phishing". This system's goal is to create an environment in which data can be collected about an institution, and effectively respond to active phishing campaigns against them. It will do so by accepting configurations that will determine the structure of how usernames are formed in their organization (e.g. the University of Oregon typically assigns usernames with a person's first name appended by the first letter of their last initial) as well as the requirements for their passwords. This information will be used to generate fake, yet realistic looking credentials to flood the phishing sites. This system will use the following structure and set of programs to achieve parts of that goal. A full fledged system will require a more thorough set of scripts and fail safes. This serves as a starting template.

II. System Design

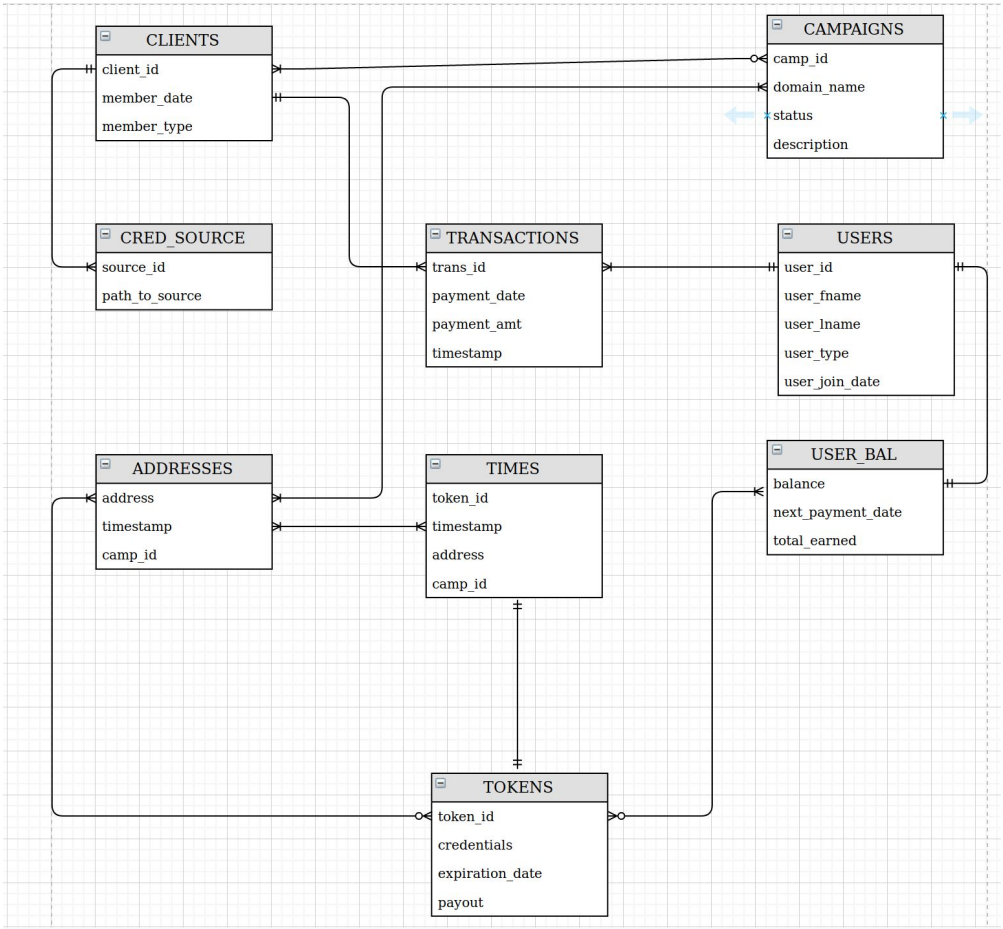


The arrows represent the flow of data from one entity of the structure to another. An example would be, The HDX_DB sends data to the HDX_API and also receives input to be stored from the API. The entity with the most connections is the HDX_API. Essentially all the work gets done via some aspect of the

API. The red lines mean a terminating connection. Meaning, that this data has been fully processed and will be sent to the phishing site. Which is the ultimate goal of this system.

The **HDX_API** will do most of the heavy lifting, as it should. It represents the API that will handle all of the RESTful requests on the data. All actions that aim to add, modify, or delete data within this ecosystem, will need to go through the API. The **HDX_DB** is another heavy hitter in this setup. The database will house the nine or so, tables that will serve as the backbone for the entire system. All permanently and most temporary data will be stored in the database for future use. The **HDX_GEN** entity will consist of the host of python or other scripts, that will house the code for creating new entries and credentials. These functions will primarily be called by the **HDX_AUTO** as well as the **HDX_WEB** applications. These two applications will be the conductors, that determine where the produced code gets sent out into the world. For the HDX_AUTO, the credentials will automatically be paired with an afflicted institution, as well as their active phishing campaigns. From here, the auto will begin automatically sending these phishing credentials at random intervals, to web pages that allow computer submissions. HDX_WEB doles out credentials to the users, which will track when the tokens actually get used and who they were assigned to.

III. Database Architecture



The database will be broken into nine main entities:

ADDRESSES - will track the IP addresses of nodes that have been used to attack phishing sites that have been added to CAMPAIGNS. This table will be used to orchestrate and track which IP addresses of users and local machines that have already been used and when they have been used to attack sites. This will allow for more data to create a more sophisticated algorithm to distribute credentials.
Primary Key: **address**
Foreign Key In: TOKENS, TIMES, CAMPAIGNS

CAMPAIGNS - will track the active campaigns against each institution. Each campaign will have a list of known IP addresses associated with the campaign. One institute can have many active campaigns against it. And one campaign can be acting against multiple institutions. This is why they have a many-to-many relationship represented in the diagram.
Primary Key: **camp_id**
Foreign Key In: CLIENTS, ADDRESSES

CLIENTS - will keep track of each institution that has signed up for the Humboldt project. It is key to note that each client (institution) will be tasked to submit their requirements for their credentials. In order to create realistic data to be submitted to sites. CLIENTS will have a one-to-many relationship with CRED_SOURCE.
Primary Key: **client_id**
Foreign Key In: CAMPAIGNS, CRED_SOURCE, TRANSACTIONS

CRED_SOURCE - will host the configurations to be used in generating credentials. One client can have many different configurations which may result in multiple CRED_SOURCE entities being attached to one CLIENT entity. This constitutes a one-to-many relationship.
Primary Key: **source_id**
Foreign Key In: None

TOKENS - will keep track of credentials, how much should be paid to a user and when it needs to be used. Tokens will be generated and paired with the address of a user as well as the user itself. This will help keep track of which users submitted which tokens.
Primary Key: **token_id**
Foreign Key In: TIMES

TIMES - tracks times of token assignments and their usage. Work on this table is imperative in creating a system that can mimic a flood of real user input to phishing campaigns.
Primary Key: **None (weak entity)**
Foreign Key In: ADDRESSES, TOKENS

TRANSACTIONS - tracks the amounts, times, and dates that institutions make their payments and users are paid. This entity is crucial in encouraging users to actually submit the generated credentials in a timely manner/at all. It is heavily reliant on the USERS and CLIENTS entities. It is in charge of tracking when a client has paid and when a user has been paid.
Primary Key: **trans_id**
Foreign Key In: CLIENTS, USERS

USERS - tracks user data. All pertinent information about a user should be stored in this portion of the database. The users are a key part of the crowd sourcing portion of the ecosystem. More attributes may be needed in the future, I simply added the most pertinent information based on my assumptions.
Primary Key: **user_id**
Foreign Key In: TRANSACTIONS, USER_BAL

USER_BAL - tracks user balances to be paid out.
Primary Key: **None (weak entity)**
Foreign Key In: USERS

Programmer's Guide

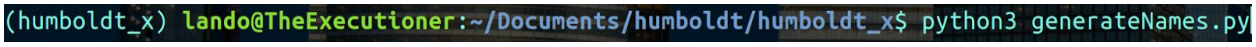
generateNames.py

This script relies on two dependencies, the json and requests imports. It consists of only one function. The overall purpose of this script is to generate random names and save them to a comma separated file in the same directory. It does so by calling the generate function. Which then returns a json response from an api call. This response is broken using json.loads() so the inner data can be extracted. The data extracted are collections of randomly generated personal information. Which is where we can grab the name and surname fields. We use these two randomly generated first and last names to write to our comma separated file, called "generatedNames.csv".

generate():

This function can be called with no parameters. Its main purpose is to simply gather the url where the api can be reached. Inside the **api_url_base** variable, we store the url that will host the query and the request to the api. We set amount equal to 500 ("?amount=500"), which is the maximum, and it returns 500 randomly generated identities. We set the region to United States ("?region=united+states"), which restricts the names to contain only letters that can be found in the English language. Which will prove to be quite useful when attempting to convince phishers of authenticity. Inside the **response** variable, we store the response we receive after executing the command "requests.get(api_irl_base)". If the request was successful, then we return the response variable containing the package. Else, we return a None type.

Example of a call:



generateCreds.py

The first line of the "MAIN" part of this script, a connection to the database "hdx.db" is created, called **con**. If sqlite cannot find an hdx.db in the same directory, then it creates a new database file with the name "hdx.db" in side the current working directory. The variable **cursor** is an instance of a cursor of type con. Cursor acts as the actual object that performs all of the commands on the tables and the database. The first execute command essentially creates a table called credentials in the hdx database, if and only if it does not already exist. If it does not exist, then it passes in the constraints, defines the attributes of the table, and then creates it.

allnames stores the name/path to the .csv file you would like to use to generate user credentials
passwords stores the name/path to the file that holds the random passwords that were generated

configuration stores the type of configuration you would like the usernames to be created from

generateUsers(*cursor, allnames, passwords, configuration***):**
This function takes four parameters, all of which are used at some point during the code block. Some are passed to another function called “generator”. Inside generateUsers, the main goal is to use the information passed in, to open the passwords file as well as the names file. While the names file is opened, each line is read, using the file.readline() call, inside a while loop. Each iteration of the loop passes in the comma separated line into the function “generator”. Generator returns a list type variable, **pair**, that holds the username created, as well as the first and last name of the user. It then adds a row to the hdx database.

generator(*cursor, name.lower(), configuration***):**
This is the function that actually utilizes the configuration originally provided in the calling of the python script. The firsts few lines are a redundancy and empty string check. Some lines that are only new line characters can cause some issues if allowed to be processed by the function. If one is encountered, a None type is returned. The **fullname** variable is a list type that holds the first name and the last name of the user. The SELECT statement does a query against the database to check if the proposed person has already been added to the database. An example wou;d be, an organization has two John Smiths. If the user does not already exist, then no added steps are necessary. We enter the if code block and create a username based on the configuration type. If instead, the query returns an instance, then we enter the Else code block. Where we create a new **count** variable. It keeps a count of users/iterations. We use this number to add onto the proposed username to ensure that each person has a unique username. Using our example from earlier the first John Smith could have username “jsmith” and the second user in the database would have the username “jsmith0” or “jsmith1” depending on how you want to implement the count.

Example of a call:

hdx.db

Sqlite3 is a fairly lightweight yet robust database management system that allows you to ship the database with your code, to the client. This is only used for proof of concept. In actual development, a more robust SQL database would be used. Examples can range from Oracle to MySQL. This particular database so far, only stores the entries of each username created thus far. Eventually, there will be multiple tables that will hold many different attributes related to the time and usage data of the generated credentials. Each entry must be unique before it can be

entered into the database. This constraint is excluding the unique identifier. Meaning, no two entries can have the same username, password, first name, and last name.

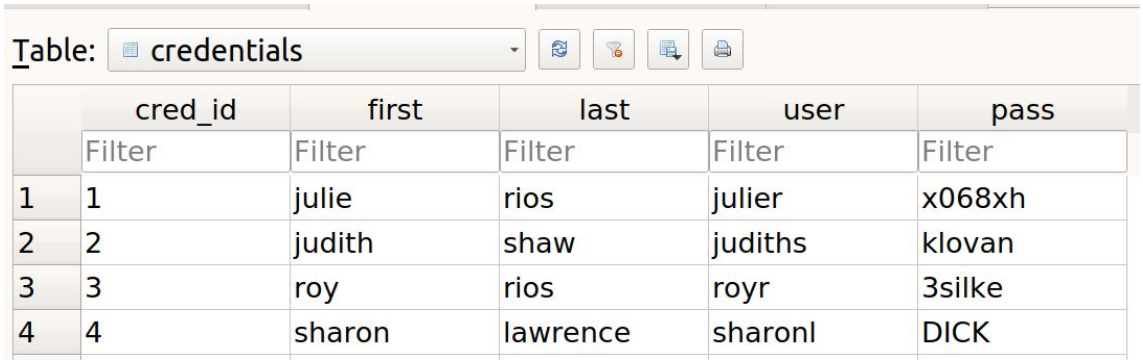


Table: credentials

	cred_id	first	last	user	pass
	Filter	Filter	Filter	Filter	Filter
1	1	julie	rios	julier	x068xh
2	2	judith	shaw	judiths	klovan
3	3	roy	rios	royr	3silke
4	4	sharon	lawrence	sharonl	DICK

Passwords.txt

This is a fairly simple file. It holds a list of ten million commonly found passwords on the internet. This file is referenced when creating the users. A random line is selected and paired with a username each call. This file gets generated by calling the API associated with the “uinames” domain. For more information on the requirements of the request, please refer to the generateNames.py portion of this document.