GRS CS 655: GENI Mini Project Final Report: Password Cracker (Fall 2021)

Manish Patel - mpatel27@bu.edu

Due Wednesday December 8, 2021

1 Repository/Project Set-Up

The GitHub repo I created for this project is at `https://github.com/mspatel927/cs655_passwordcracker`. The RSpec file used to set-up this project on GENI is located in the root directory, entitled "PasswordCracker_request_rspec.xml". A screenshot of the resources ready is also located here, under the name "resources_ready.PNG". The code that is used to implement this project is all located in the src directory of the repo. Within src, the master-node directory contains the code relevant to the the master node of our set-up that takes in password cracking requests (i.e. from the web interface) and delegates work to the worker nodes to crack the password. The webserver directory within src contains the code for running the web interface at which users can make requests to crack a MD5 hash. Also, the worker-node directory here holds the code that the worker nodes use to take in work delegated to them and using a brute-force method, look for the de-hashed password. Regarding more on GENI, there is a slice on GENI created under the CS-655-Fall2021 project entitled PasswordCracker that holds the resources for this project (slices and resources set to expire on 12-25-21, nodes all with Publicly Routable IP's). The slice can be found at `https://portal.geni.net/secure/slice.php?slice_id=3fd72465-f990-4ef0-9e20-592779e2a61c`.

2 Introduction/Problem Statement

This project, as the title above suggests, is focused on the Password Cracker problem, where we want to be able to "de-crypt" a password given it's MD5 hash. Given such an MD5 hash made up of 5 alphabetic characters, the objective is to be able to figure out what the original password that was hashed was. More information can be found at `https://www.md5hashgenerator.com/` in order to see how a given 5-character string would be translated into an MD5 hash. There is no way to actually de-crypt a hashed password, because MD5 only works in one way. Therefore, the only approach is to do so by brute-force and iterate through every possible 5-character alphabetic password, find the MD5 hash of the password, and see if it matches the submitted hash. As many are, brute-force algorithms can be incredibly time expensive so one of the more efficient ways to do this would be in a distributed or multi-threaded manner. Assuming we use the 26 uppercase characters and the 26 lowercase characters, each of the 5 characters of the input has 52 possibilities, leaving the system to iterate through $52^5 = 380204032$ possible passwords.
With all this, the goal of the project is to create an architecture to allow a user to submit such an MD5 hash to a web page and after being "worked on" by the nodes on the back-end in a distributed way, the original password should be returned once it is found by a node. The learning outcome here is to figure out how to build the connections between such a web page, an intermediate master node, and a set of multiple worker nodes that would allow all of them to communicate together to crack the MD5 hash. For one, each of the worker nodes need to be able to work together so that they are each trying different sets of passwords and see if they match the input but also stop their processes once one of the nodes has cracked it. Additionally, the web page and master node have to communicate to take the input and return the cracked password. Lastly, the master node has to communicate with the worker nodes to delegate the work to each of them properly. Therefore, the purpose of this project is to figure out how to properly create each of these "lines" of communication and implement the distributed nature of the project so that experimentation can be done to to compare different versions of this architecture.

3 Experimental Methodology

The architecture of this project revolves around 3 distinct "sections." The first part is the web interface, run with the Flask framework written in Python, where the user actually interacts with the entire project. This is where the user inputs the hash that they wish to crack and also specify the number of nodes that they want to distribute the "cracking" on. Then, once they submit, the hash and the number of nodes is sent to an intermediate master node. This node, after receiving these two pieces of data, initializes socket connections with the appropriate number of worker nodes and sends them all a copy of the hash that needs to be cracked. It splits up the workload between the nodes and after sending the hash out, it awaits a response from one of them with the cracked password. At this point, the worker nodes are each running brute-force algorithms to iterate through different combinations of the lowercase and uppercase alphabet (starting at different points in order to cover more ground for more nodes). In other words, they use the itertools package of Python to go through a list of all of the letters and one-by-one, create combinations in order of the letters in groups of 5. The starting letter in the sequence determines where the algorithm starts its iterations so changing the order of the inputted letter sequence changes which part of the alphabet the specific worker node handles. Each combination is hashed and if it matches the input hash, we have found our password. Once any of the nodes finds this match/password, it sends it back to the master node and the master node closes its connections to all worker nodes, thus ending their iterations. Then, the master node sends the found password to the web page and the web page displays it at the bottom to show the user. Each of these sections is connected through TCP, as they communicate through socket addresses assigned by GENI upon reservation of the resources. This allows the data to be reliably sent back and forth.
Each of the implementations of this project are done in Python. Thus, the workers and master node are written in Python and use the socket package to create connections between each other. In fact, the master node acts similar to a client node as it sends out a "message" (which is the hash) to the worker nodes. The workers are each like server nodes in how they take this hash, do some processing, and send back a response to the master node. The scalability aspect of this project comes with the number of worker nodes. Although not fully functional in this implementation, the user should be able to specify how many worker nodes it wants to utilize to crack the given password. This would mean that the master node needs to divide the number of alphabetic combinations of passwords appropriately between the specified number of worker nodes and send only them the hash. No other nodes should be connected to or sent the hash.
This experimentation will be under the assumption that we only have 1 user sending MD5 hashes to the web-page at a time. This means that no more than 1 person should be accessing the web interface to send requests to the master node at the same time. Additionally, based on the architecture created with GENI, there is a maximum of 5 worker nodes that can be used. Thus, although it is validated by the form on the web interface, the user can only ask for between 1 and 5 (inclusive) worker nodes to be used to crack the password. Although it cannot be validated, it is assumed that the inputted hash actually comes from a 5-character password of only lowercase and uppercase letters.

4 Results

4.1 Usage Instructions

This experiment is accessed by users through the web interface provided. Although previous iterations of the project used the command line to directly submit the hash to the master node, the final product uses the web page (hosted locally on the default Flask address) to hold the form where the user can specify the hash they want to crack and the number of worker nodes. In order to set-up the entire experiment, one must first reserve the appropriate resources and topology using the Rspec provided on GitHub. Then, on each of the nodes (master and worker), the user must run the appropriate scripts to get the python files (master.py, worker1.py, etc.) and run them so that they are up and running, ready to be connected to by the other nodes. Once they are all ready, the Flask application can then be run so that the user can submit the aforementioned information to get everything started. This workflow is demonstrated, for the most part, in the video on GitHub.

4.2 Analysis

The main aspect of this experiment that I wanted to test was how the network conditions affected how long it took to crack the password. The TCP connections between the web interface and the master node as well as that between the master node and each of the worker nodes can be affected in many different ways. The percentage of lost packets, the delay, the bandwidth, and more are the variables that can be manipulated through GENI to perform such experimentation. Time constraints did not allow me to fully test all of these variables. The numbers are not included in this report, but more visual representations of the experiment can be found on the GitHub repo. The main finding overall was that the network conditions did not have a very significant effect on the password cracking because the majority of the time was spent on the brute-force algorithm itself. Depending on the hash itself, since our algorithm goes in order starting at certain letters, the starting letter of the hash actually has a very large effect. The conditions of the TCP connections do not have a large effect because the data being sent back and forth is not very large. Thus, it is almost negligible and the response time is heavily dominated by the loops run on the worker nodes instead.

5 Conclusion

The main findings of my project overall reflect how much of a difference distributing such a workload can make. With one node, we would start iterating at the letter "a" but with multiple nodes, we can start at several points in the alphabet and thus, not have too much of a concern about passwords starting with letters later in the alphabet. Scalability and distributing the workload make such an impact and instead of the TCP network conditions having a more significant effect, it was more so about the architecture of the network and how the jobs were divided.
In its current state, the project does not allow the user to actually change the number of workers on the fly based on what is specified. However, we could extend the project to make such a change so that the number of worker nodes being used is actually set up on the spot for every user rather than staying at 5 for every trial. Additionally, there were certain troubles with connecting the web interface with the master node, likely due to routing. Despite certain successes (to some extent), re-configuring the routing can allow for less problems with this, so that the data is sent in a much more smooth manner along this connection. Lastly, dividing the workload was done in a pre-determined way here but multi-threading is a very viable solution that can allow us to more efficiently split the workload (especially with a variable number of workers) so that we can have simulatenous socket connections that can close at a much more efficient rate when the password is cracked. Other extensions of this project could include having longer passwords, alphanumeric passwords, having the user specify certain parameters (such as the bandwidth and delay), using caching to store the previously found MD5 hashes so that they do not need to be re-computed on future rounds or iterations, and more.

6 Division of Labor

This project was completed by myself. Therefore, there were no other contributing members on this project. However, I would like to acknowledge references that I used to complete this project, which included GENI tutorials (`https://groups.geni.net/geni/wiki/GENIExperimenter/Tutorials`) and pre-existing GENI Rspec files on the website to be able to set-up this project and manipulate the nodes accordingly.