CSC 347

A1: Report

Michael De Lisio & Arshia Gharai

4th October, 2019

Part A

Easy Dump

Cracking the easy_dump.txt file required less clever or intuitive methods, hence the files name. The folder "WordLists" was the result of scanning the web for a while seeking leaked databases. The bash script "easy_script.txt" contained only a few lines of code which effectively looped through each leaked password dump file and ran the command `./john --wordlist= $dumpfile --rules:All easy_dump.txt` (where "dumpfile" is the hashed file). This task completed in less than a minute further speaking to how poorly the passwords were constructed. All ten were cracked and remain in the easy_passwords.txt file.

Medium Dump

In order to crack the Meduim_dump file, we had to use a combination of rules with a couple of different word lists. I personally used some of the word lists which were in /usr/share/dict since there was a variety of different lists which made it easier to be able to crack the file. This process was very time consuming given the size of the word lists.
List of commands used:

- `./john --wordlist=/usr/share/dict/cracklib-small -rules=Jumbo medium_dump`
- `./john --wordlist=/usr/share/dict/ -rules=Jumbo medium_dump`
- `./john --wordlist=password.lst -rules=Jumbo medium_dump`
- `./john --wordlist=/usr/share/dict/american-english -rules=Jumbo medium_dump`
- `./john --wordlist=/usr/share/dict/words -rules=Jumbo medium_dump`

As you can see, I used the "Jumbo" rule for most of the operation along side of a variety of different word lists and dictionaries to be able to crack the passwords in the medium_dump.

Hard Dump

The initial approach to cracking the hard_dump.txt file was to concatenate every word in the given dictionary file ("english-small.txt"). The concatenation process was quite computationally

expensive in itself - given the length of the dictionary. It's important to first consider the feasibility of such a task; concatenating a 4976 word dictionary with itself is comparable to the problem of finding all permutations of sampling 2 items from a set of 4976 distinct items where order matters. This is essentially n^2 words (where n is the number of items in the set) which intuitively implies a doubly nested for-loop, which is exactly the implementation of the python script concat_dict.py. The program also spawns n threads directly relative to CPU cores of the machine, helping expedite the task.

The "divide and conquer" method was also applied to the actual cracking process. After some inspection of the JtR functionality and commands provided, it was evident that the fork utility would be very useful. While using the word list "concat_dict" of 4976^2 words, four processes were forked to divide the problem over the DeerField Hall machines - which contain quad-core processors.

List of commands used:

- `./john --format=md5crypt --wordlist=listforme.txt -rules=Jumbo hard_dump` {listforme.txt is a leaked list of passwords that I found @ https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-100000.txt}
- `./john --format=md5crypt --wordlist=/usr/share/dict/cracklib-small -rules=Jumbo hard_dump`

Salted Password Hashes

Salting a password is a method of making it computationally harder to decrypt. A salt is a string of bits randomly generated which is then combined with the plaintext password and hashed. Adding a salt of k-bits to a hash adds 2^k extra permutations of password possibilities making the effort of computing its correct decryption much more computationally expensive. To crack the dumps above dumps which were md5crypt salted password hashes, the "--format=md5crpypt".

Part B

Cracking the Secret File

When decrypting text files using the openssl2john.py file and then running the "hashfile" generated from that program, the message "Note: This format may emit false positives…". This inconvenience arises from the lack of a header contained in these generic ASCII encoded text files that are fed into the ./john program. File formats such as PDF, zip, RAR, etc. have predefined constants located in their file format that can be reliably checked by ./john so that it is clear when the file has been successfully decrypted.

A special argument provided by the john binay was employed to avoid the frustrating task that is checking for every false positive that was generated. The exact command entered to generate the hashfile was: `./openssl2john.py -c 0 -m 2 -a 100 secret_file.aes256.txt`. The "m" option indicates that the file is hashed with SHA256, the "c" option declares the cipher type which in this case was aes-256-cbc and last the "a" option defines the percentage of the encrypted file that is encoded with ASCII characters. The bottom of the ./openssl2john.py summarizes these options best. The "a" option was vital to eliminating every false positive, given the important detail from the assignment handout which stated the entirety of the text was ASCII encoded.

Every file used to successfully decrypt the file "secret_file.aes256.txt" is located in the Part B folder of the extras directory of the git repo. The main script was the "aes-cbc-script.sh" which effectively hashed the secret file then ran the ./john binary on a word list (also located in the same directory) which used the "All" rules option and the format of "openssl-enc". The output of the john binary was piped into a bash routine named "process_candidates" which processed every possible candidate password generated by john and tested them with the openssl decryption option. The script does not terminate upon receiving a candidate instead it logs it in a file named "report.txt". Note that the path to the john binary is specific to the personal MacBook rather than the DeerField Hall lab machines. It took roughly 30 seconds to find the correct password.

Cracking the RAR File

The RAR file took an exceptionally long period of time to crack compared to the hash dumps and the aes-256-cbc file encrypted using openssl. This is because of the expensive RAR encryption routine which is specifically designed to be slow to deter hackers from running brute force attacks and dictionary attacks. It is estimated that the RAR encryption routine can take 210 years to run every permutation of an 8 character printable word [1].

To crack the secret_song.rar archive we used the same pattern of the secret_file.aes256.txt file. Observing the construction of the secret_file.aes256.txt password implies the following rule: capitalize the first letter and append some combination of digits. The "myrarrule" JtR wordlist rule in the "john-local.conf" file was called in the "rar_script.sh" with the command: ~/JtR/run/./john --wordlist=$list --rules:myrarrule --fork=2 rarhash. This command was run in a for loop that iterated over a directory of wordlists employing the custom myrarrule rule. The wordlist file that was used on the successful iteration was a dictionary file entitled "cracklib-small.txt" taken from the DeerField Hall lab machines /virtual directory. It took approximately 65 minutes until the John the Ripper cracked the password, which can be found in the a1/rar_password.txt file.

Source:

[1]. "Why Is It so Hard to Crack RAR Passwords?" *UnRAR-Crack Benchmark - Technical Q&A*, 2011, anrieff.net/ucbench/technical_qna.html.

Part C

Description of how the find_username.py works:

First thing done in the script is declaring the headers of the website which was accessed through the inspect feature of firefox. Next, we open the text file which in this case was the facebook database and run .strip on it so we just access the word itself. Next, we write a loop where in goes through all the words in the database. Next, we declare a user, an address and a proxy which are the usernames that were in the database, the URL and the proxy SOCKS5 which allowed us to access the website from outside the campus respectively. Then we use a request.post method which allows us to access the website and check the usernames to see if they are active or not. Last but not least, we check the status we get from the request.post to see if we got an active username or not and then we close the facebook database text file that we opened in the beginning.

Description of how the crack_accounts.sh works:

The crack_accounts.sh shell script utilizes the curl command line tool to post username and password data to the URL http://142.1.44.135:8000/login. The usernames to be used is based on a file containing usernames that were found using the find_username.py script. The crack_accounts.sh script iterates through this file of valid usernames and also requires a dictionary of potential passwords, which it also iterates through and tests on each username. The main utility used for testing authentication was the following: curl --socks5 127.0.0.1 -u $username:$password 142.1.44.135:8000/login ($ indicates a variable in bash). The script compares the html output from this command with a HTML page of an invalid account using the "diff" command. If the diff command produces output then it must mean that the curl commands output is of a valid password. When the script found an active account it printed a message to standard output. When running the command on a machine off campus we used the proxy option: socks5 that effectively rerouted our connected through the DeerField Hall machines granting us access to the website.

Cracking strategy:

Our cracking strategy was to first find valid usernames. We would go through every username in the facebook first names database and try them on the website. After this step, we check the HTML file that was generated from a given curl command and compare it to a reference HTML file of a invalid username. The comparison was done using the diff command line utility, the logic being, if the diff command output data then the two HTML files must be different implying the username may not have been rejected. Though, the curl command gave us an incredible amount of issues. There were constant “bad request 400” returns and often when trying the curl command against usernames which we had known to be valid already the command would produce either error or the same login HTML file. Our first script entitled websites_script.sh was our first attempt at solving this problem (its located in the extras directory).

We then decided to write a python script which effectively did the same thing that the websites_script.sh intended to accomplish. We imported the requests python module to make HTTP posts to the websites server. Once acquiring a few usernames from the script we used the crack_accounts.sh shell script which used curl to authenticate username and password pairs from a word list bank. This script takes two arguments; first being a username bank of valid usernames and second a path to a word list file to check passwords with.