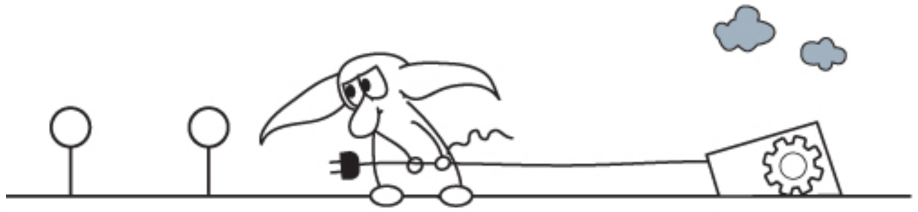
Cracking 400,000 Passwords Readme



Message from the Author

Let me first start by thanking you for showing an interest in this research. It's been a weird two years since I decided to quit my job and go back for my PhD in Computer Security. I remember sitting outside complaining about school with a couple of the other graduate students shortly after arriving at Florida State. One of them turned to me and said, "Well you have experince programming so this class should be fairly easy for you." "Naw" I replied, "I haven't programmed more than a shell script in years." "At least you have the Math background which should help on this project" he continued on. "Nope, I'm horrible at Math. I have no idea how I got through my undergrad with it." He thought for a second and then said, "Ah, so you're more of an Algorithms and Theory person. A couple of professors have been looking for someone like that." I laughed a bit and replied, "No, algorithms was my worst subject." He stared back at me. "What the Hell are you doing here then?!"

I think I'm finally closer to finding the answer to that question, but my original statements remain valid. My code has bugs, my proofs have weaknesses, and my programs run slow. That being said, I want to share what I've done as part of my investigation into the art of password cracking. At times it feels more like a rediscovery as I'm sure much of what I am learning has been done before. I hope this helps you or at least provides inspiration for you to develop your own tools. Included in this packet are the tools I could get somewhat presentable by the submission deadline. Like most deadlines, this one crept up on me, Ninja style and struck without warning. If anything interests you I highly implore you to check out my website where hopefully in the time between now and Defcon, bugs will be beaten, promised features will be implemented, and additional documentation will be written. Or at least there will be a post explaining how the laziness pirate hijacked my coding ship.

Matt Weir
E-Mail: weir@cs.fsu.edu
Webpage: http://reusablesec.blogspot.com

John the Ripper Config Files

I attached a few of the standard config files I use with John the Ripper. I will attempt to post more to my website. I've been fairly bad at actually saving my configs since I generally construct one or two rules, run them for a while, and then write over/modify them for the next attack I want to run. I do want to state that the default rule set that comes with John the Ripper is extremely effective against simple passwords. The only downside was that it was written to complete quickly, making less than 38 million guesses when fed a dictionary containing around a million words. Thus, unless the words are pre-mangled in the input dictionary it will not crack "strong" passwords. For those, you need to craft the rules yourself.

john-modified.conf

Some standard rules that I've found effective. Contains single letter replacements, numbers added to the end, well you get the idea.

basic-rules.conf

These were the rules that I used in my default dictionary based rainbow tables for drcrack. They actually are in reverse order since drcrack is slightly more efficient if the larger rules are near the beginning. I also include basic-rules.load which is the jtrMakeConfig rules file if you wish to load them and modify them using jtrMakeConfig.

john-bruteforce.config

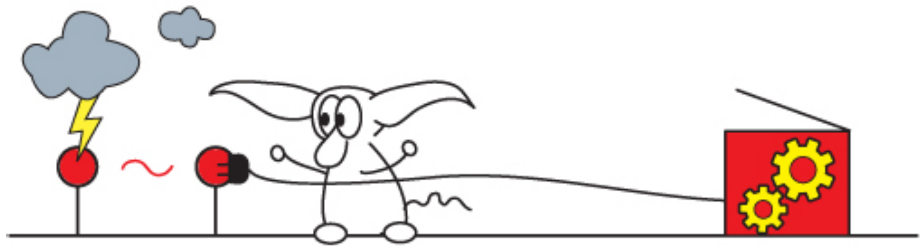
These rules are for running a targeted brute force attack in John. To use them, just feed in an input wordlist containing the lowercase alphabet, with one character per line. Nowadays you can do much the same thing easier by piping the output of Crunch directly into John the Ripper.

Rules by Other People

These were posted to the John the Ripper Mailing list by Minga. All I want to do is highlight them for your use, and all credit should go to him for the work he has done.

http://marc.info/?l=john-users&m=123820850908275&w=2

http://marc.info/?l=john-users&m=124053430313891&w=2



Probabilistic Password Cracker

Overview

This is a major area of research for me, and something that I truly believe in. For years, we have been applying probability models to help speed up brute force attacks, (aka letter frequency analysis and Markov Models). At the same time though, our approach to dictionary based attacks has been fairly ad-hoc. John the Ripper's default, (and single mode), rules while built based on their creators experiences with cracking passwords, are still extremely subjective. For example I've found very few passwords in my cracking attacks that were created by reversing an input dictionary word. Cain and Able, while a great product, probably has the most bizarre rule selection in that it focuses on capitalization mangling at the expense of just about everything else, (though it will also add two numbers to the end of words and replace letters with numbers). AccessData orders their default rule set not on how effective the rules are but by how large the search space is for each rule. This is not a slam on these approaches but I do think that as passwords become stronger and stronger, (either through user training or password creation policies), we need to improve how we generate and use word mangling rules.

The main goal of this project is to see if we can assign probabilities to different word mangling rules and then generate password guesses in probability order. There are several advantages I feel this approach offers us. First, by designing a way to measure the probability of word mangling rules, we can quickly generate new rules by training our password cracker on known passwords that we feel are similar to the target. This way, we will be able to train our cracker to go against English speakers, Russian speakers, passwords created for social networking sites, passwords created with a strong password creation strategy, etc. If you've ever spent time editing a John the Ripper config file, you know that ability to automatically generate rules is very nice. Second, it allows us to more effectively target strong passwords. Just like with letter frequency analysis, the letter "z" may be uncommon, but the string "aaaaz" may be more probable than the string "dfttp" since it takes into account the probability of all the letters. Likewise, by using a probability model of how passwords are created, we can better balance the order of how multiple word mangling rules are applied to password guesses. For example, the guess "$$password63" may be more probable than "!*password12". Not only does this technique apply to word mangling rules, but also to the input words themselves. We know that the word "password" is more probable than the word "zebra". Using a probabilistic approach gives us a framework to make use of this knowledge.

Special Thanks and Acknowledgments:

The original version of the program was written Bill Glodek, another graduate student at Florida State University. The original idea for using probabilistic context free grammars to represent how people create passwords was Dr. Sudhir Aggarwal's and Professor Breno de Medeiros's. Basically I was lucky enough to come in at the right time to assist with the start of the program and help carry it on once Bill graduated.

Theory, and Experimental Results:

For the theory behind our approach, an overview of the basic algorithm, and some of the results of using an early version of our password cracker against real passwords please see the included PDF of the paper we presented at the IEEE Security & Privacy conference, "Cracking Passwords Using a Probabilistic Context Free Grammar." I'll try to post additional results to the website, but the short preview is that the current version, (as feature incomplete as it is), helped out quite a bit with cracking the phpBB list and was one of the main tools that I used.

Cracking 400,000 Passwords Readme

To Do List:

The version included on the CD is still very much in the Beta phase since I'm right in the middle of completely rewriting the training program to add new features. The main limitations include

1) The training program has "issues" when being trained on passwords that include non-ASCII characters. There's a workaround, but it's really messy.

2) Currently the training program doesn't learn case information from the training password set

3) The training program doesn't learn letter replacement rules from the training password set

4) Need to integrate CUPPS support, (from the remote-exploit team), so we can better make use of collected information, (birthdays/children names/etc), in our password cracker.

5) It would be nice to integrate dictionary evaluation into the training program instead of having to use a different program, (aka passPattern).

6) Add support so the trainer can create multiple named rulesets automatically, (rather than overwriting the last ruleset forcing the user to manually back it up).

7) General performance improvements in the password guess generator itself

8) Saving session state information from the guess generator so it can be stopped/restarted

9) Add the ability to detect user input and output the current guess/status to stderr

Use and operation: TRAINING:

The version on the DVD is already trained on the MySpace password set which we've found has been fairly effective, (this was the list of MySpace passwords that were stolen by phishers several years ago). Please note, when you train the password cracker on a new list it will overwrite the previous rules. To train the password cracker on a new list, just run

./process.py <training list>

The training list should be a newline separated file of the raw password values. Aka

password1

password2

2password

....

Use and operation: CREATING PASSWORD GUESSES:

First you need to build the password cracker from the source files. It has been tested on Ubuntu Desktop, and MacOSX, (with Xcode installed). To build the executable, simple type:

./make

Once the executable has compiled, (hopefully without errors), it is ready to run. Currently it is fairly stupid, (I'll blame the program vs. my programming skills), so it needs to be run from the directory it is installed in, (I need to strip the path info off from the command line and use it in future versions). Also, the training grammar, (aka rules), need to be installed in the same directory as well. You should have three folders, "**./grammar**", "**./digits**", and "**./special**" that represent the full training grammar. These folders will be filled with various files labeled such as "**1.txt**" or "**structures.txt**" which contain the probability information gathered from the passwords it was trained upon.

To run the program type

./pcfg_manager <options>

Options:

-dname[0-9] <dictionary name>	**<REQUIRED>: The input dictionary name** **Example: -dname0 common_words.txt**
-dprob[0-9] <dictionary probability>	**<OPTIONAL>: The input dictionary's probability. If not specified set to 1.0** **Example: -dprob0 0.75**
-removeUpper	**<OPTIONAL>: don't use words from the input dictionary that have uppercase chars**
-removeSpecial	**<OPTIONAL>: don't use words from the input dictionary that have special chars**
-removeDigits	**<OPTIONAL>: don't use words from the input dictionary that have digits**

Generally I've found it to be more effective to use the -remove options and let the program, vs the input dictionary determine where to use special characters and digits. The one exception is the -removeUpper option which can allow you to attack case mangled words by using an input dictionary that contains words where different case mangling rules have already applied to them.

The -dprob option ranges from 1.0 to 0.00, and should be set based on what percentage of the passwords you expect the input dictionary to crack. For example a very large input dictionary might have a probability of 0.78. Note, the program is smart enough to assign a final probability by taking into account the probability set by the -dprob option and the size of the input dictionary. This way, even though a very small but highly effective input dictionary might have a -dprob setting of 0.05, and a much larger input dictionary might have a -dprob setting of 0.40, the smaller dictionary will be used more often. Also, duplicate dictionary words will be removed from multiple dictionaries, with the duplicated word being assigned to the dictionary with the highest final probability. For example, the word "football" may show up in several of the input dictionaries that are used, but the password guess generator will place it in the most probable input dictionary to maximize the amount of times it is used.

An additional program for evaluating the probability of input dictionaries:

I included an additional program, **passPattern**, which measures the effectiveness of input dictionaries by using the idea of edit-distance. It can be used in conjunction with our probabilistic password cracker to assign a value for the -dprob option. In addition, this program creates a golden dictionary of all the words found that would have cracked passwords in the training set which allows you to combine several input dictionaries into a single best of breed dictionary. Also it stores all passwords that would not have been cracked in the file "unclassified.txt." I have found this very useful for narrowing down the number of passwords I need to manually inspect when looking for new mangling rules. Finally, it records the word mangling rules used and how often they were used in the file "editFile.txt". To run this program, first use the make option to compile it and then type

```
./passPattern <dictionary> <password list>
```

The options should be fairly self explanatory.

Evaluating Different Password Lists

It is often helpful to be able to quickly evaluate a list of cracked passwords both for information to improve future cracking sessions and to distribute information about how people create passwords without revealing the actual passwords themselves. One useful program is passPattern. As stated previously it will record how often different word mangling rules occur. Furthermore, you can run it using targeted dictionaries, such as "Sports Team Names", to see how many people used those base words.

Another approach is to use the program **pass_stat** which will record common statistics about a password list such as average password length, letter frequency analysis, number of passwords that contain uppercase characters, etc. To run it, first use the **make** option. Once it is compiled, just run

./pass_stat -file <list of cracked passwords> -totalPasswords <total number of passwords>

The -totalPasswords switch is optional, but nice since unless you are dealing with a plaintext list, chances are you have not cracked all of the passwords. When that option is used, pass_stat will print out some additional statistics of what the numbers would look like if the uncracked passwords followed certain patterns.

I'm not really happy with the average complexity measurement and I would appreciate any additional input on it. All it does is give passwords a score based on how many criteria they meet.

-Contains at least one lowercase letter: +1

-Contains at least one uppercase letter: +1

-Contains at least one special character: +1

-Contains at least one digit: +1

-Contains at least one non-ASCII character: +1

-Is at least eight characters long: +1

Therefore, the password, "monkey123" would have a complexity rating of 3, (longer than eight characters long, and contains lowercase letters and digits). Any non-blank password will score a value of at least 1, and the maximum complexity score would be 6. An average complexity rating of 2 would imply that on average one mangling rule was being applied to the passwords, (though of course there are exceptions, such as the password "password", or "12abc12" which both have a complexity of 2).

Included in the DVD are the results of running pass_stat on both the cracked passwords from the phpBB.com list and on the cracked passwords from the Finnish78k list, (I only attacked the unsalted MD5 hashes)