A Practical Guide to Cracking Password Hashes

This post is the first in a series of posts on a "A Practical Guide to Cracking Password Hashes". Cracking passwords is an important part of penetration testing, in both acquiring and escalating privileges. The aim of this series is to describe some of the techniques that MWR has found to be effective at cracking both enterprise level and consumer passwords.

In addition to the techniques covered in this post, the series will cover:

- Contextual Attacks
- Combinator Attacks
- Topographical Attacks
- Markov Attacks
- Dynamic Attacks

This post will focus on rule based attacks against passwords. It will explain why in general, opting for a targeted more efficient ruleset over increasingly large dictionaries can yield better results. Furthermore, this post will describe how to write password cracking rules and test these rules empirically.

Tools

There are three sets of tools MWR uses to crack passwords. This list is not comprehensive, but covers most of the password cracking that MWR does. The tools are:

- Hashcat[1]
- Hashcat Utils[2]
- PACK[3]

Hashcat is a tool that provides an extremely efficient way to convert plain text collections of characters into their hashed equivalent. Hashcat is free, but the development team headed up by Jens 'atom' Steube has decided to keep the code base for it proprietary. Hashcat utils contains a number of tools that allow for some more complex and interesting techniques to be used to crack passwords that will be discussed in later posts in the series. PACK is a set of tools developed by Peter Kacherginsky to perform analysis on sets of cracked passwords and use this analysis in attacking password hashes in the future. There are a number of alternative password cracking tools available, such as John The Ripper that can be used in similar ways, however, hashcat exists as the mainstay of MWR's password cracking arsenal. Either tool can be used in following along with this series, although if it does interest you, you can take a look at discussions such as this one with regard to the performance of each tool[4].

Getting Started

Wordlists and rules are, in many cases, the backbone of a password crackers attack against passwords. Wordlists are readily available online, but the best wordlists are typically one's that are developed and tuned over time by a password cracker.

In order to follow along with the series, download the Battlefield password hashes from here[5]. The wordlist that will be used throughout the series is `phpbb.txt` which is available here[6]. In the interests of simplicity, a single wordlist will be used. The Battlefield Heroes website was compromised in 2013. The attackers gained access to the Battlefield Heroes database which contained user profile information including usernames and password hashes. This information was subsequently published online by the attacker(s). Throughout the series, these leaked MD5 hashes are going to be used to practice against as a case study to practice the techniques discussed in this series. Since then, the users of the website have had to change their passwords and the password hashes are not associated with user accounts. This allows us to develop techniques against real world list of passwords that does not put the users of the application at risk. A good candidate for this wordlist is `phpbb.txt` as it contains a number of common passwords and is relatively small in size.

Wordlist, Go!

Before adding rules to the attack, an attack against the hashes using solely the wordlist can be performed. After downloading the wordlist, password hashes and hashcat, a simple attack can be launched using the following command (assuming a 64bit architecture is being used):

```
./hashcat-cli64.bin -m 0 bfield.hash phpbb.txt -o plain_wordlist_results.txt
```

The `-m` flag informs hashcat of which hashing algorithm to use. A comprehensive list of all of the algorithms that hashcat supports can be obtained by running `./hashcat-cli64.bin --help`. The next argument should be the location of the wordlist that is to be used. The `-o` flag tells hashcat to write the results of an attack to a file. After running hashcat against the battlefield hashes the results shown below should be obtained.

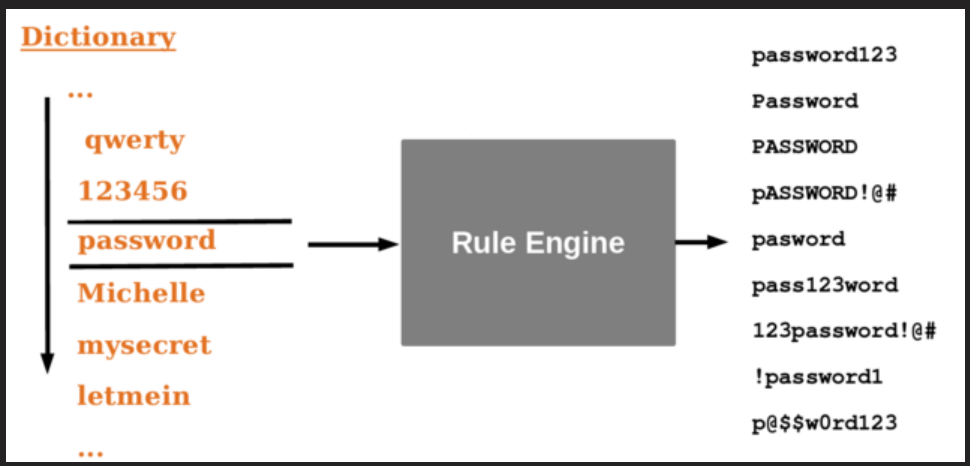
```
Initializing hashcat v0.49 with 8 threads and 32mb segment-size...

Added hashes from file ../UniPresentations/bfield.hash: 548686 (1 salts)

NOTE: press enter for status-screen


Input.Mode: Dict (../Dictionaries/GeneralLists/phpbb.txt)
Index.....: 1/1 (segment), 184389 (words), 1574395 (bytes)
Recovered.: 22542/548686 hashes, 0/1 salts
Speed/sec.: 494.74k plains, 494.74k words
Progress..: 184389/184389 (100.00%)
Running...: --:--:--:--
Estimated.: --:--:--:--

Started: Fri Jul 17 10:45:53 2015
Stopped: Fri Jul 17 10:45:55 2015
```

The wordlist has successfully cracked 22542/548686 hashes (4%) of battlefield in a few of seconds. After obtaining promising results with a relatively small dictionary, a natural next step perhaps is to increase the size of the wordlist and re-run the same attack. While this is an option and in many cases a viable next step, an alternative approach is to introduce rules into the attack. Using intelligent rules instead of simply increasing the size of the wordlist used to crack passwords has a number of benefits:

- Larger wordlists require more diskspace. In the event that 1000 rules are applied to a wordlist, the number of password candidates generated would be the size of the wordlist x 1000. In the event that a 1GB wordlist was used, in order to represent the same number of password candidates without rules would require the size of the wordlist to be approximately 1GB x 1000 = 1TB disk space.
- Rules used in conjunction with wordlists are generally faster than using a large wordlist that contained all of the candidates that would be produced by a rule based attack. This is because of the I/O overhead involved with using a large wordlist. By using smaller wordlists and rules, it is possible to generate a significant number of password candidates more efficiently.
- Wordlists often already contain the base or root word of a password that needs to be guessed. This word simply needs the right manipulation performed on it in order to produce the correct password candidate. Therefore, it is possible to limit the redundancy in a wordlist by maintaining a list that contains important root words and supplementing these with rules.

Rules

To perform a rule based attack, a set of rules and a wordlist are required. A rule defines some kind of preprocessing that each word in a given wordlist will undergo before being hashed. An example of a rule might be to append the characters `123` to the end of each password candidate that is generated. Alternatively, a rule might substitute all occurrences of the letter `a` with the `@` character. By compiling a list of rules that emulate such user behavior, a simple wordlist can be turned into a highly effective one by generating variations on words that produce other likely password candidates. This process is diagrammatically represented below:



Hashcat iterates through a list of words and feeds each one through its rule engine. The rule engine reads in a specified rule file and manipulates the input word according to each rule. Hashcat has a language for defining rules to be used with wordlists. They allow for some quite complex manipulation of words. The following image is taken from the hashcat website[7].

Name	Function	Description	Example Rule	Input Word	Output Word	Note
Nothing	:	do nothing	:	p@ssW0rd	p@ssW0rd	
Lowercase	l	Lowercase all letters	l	p@ssW0rd	p@ssw0rd	
Uppercase	u	Uppercase all letters	u	p@ssW0rd	P@SSW0RD	
Capitalize	c	Capitalize the first letter and lower the rest	c	p@ssW0rd	P@ssw0rd	
Invert Capitalize	C	Lowercase first found character, uppercase the rest	C	p@ssW0rd	p@SSW0RD	
Toggle Case	t	Toggle the case of all characters in word.	t	p@ssW0rd	P@SSw0RD	
Toggle @	TN	Toggle the case of characters at position N	T3	p@ssW0rd	p@sSW0rd	*
Reverse	r	Reverse the entire word	r	p@ssW0rd	dr0Wss@p	
Duplicate	d	Duplicate entire word	d	p@ssW0rd	p@ssW0rdp@ssW0rd	
Duplicate N	pN	Append duplicated word N times	p2	p@ssW0rd	p@ssW0rdp@ssW0rdp@ssW0rd	
Reflect	f	Duplicate word reversed	f	p@ssW0rd	p@ssW0rddr0Wss@p	
Rotate Left	{	Rotates the word left.	{	p@ssW0rd	@ssW0rdp	
Rotate Right	}	Rotates the word right	}	p@ssW0rd	dp@ssW0r	
Append Character	$X	Append character X to end	$1	p@ssW0rd	p@ssW0rd1	
Prepend Character	^X	Prepend character X to front	^1	p@ssW0rd	1p@ssW0rd	
Truncate left	[	Deletes first character	[	p@ssW0rd	@ssW0rd	
Trucate right	]	Deletes last character	]	p@ssW0rd	p@assW0r	
Delete @ N	DN	Deletes character at position N	D3	p@ssW0rd	p@sW0rd	*
Delete range	xNM	Deletes M characters, starting at position N	x02	p@ssW0rd	ssW0rd	*
Insert @ N	iNX	Inserts character X at position N	i4!	p@ssW0rd	p@ss!W0rd	*
Overwrite @ N	oNX	Overwrites character at postion N with X	o3$	p@ssW0rd	p@s$W0rd	*
Truncate @ N	'N	Truncate word at position N	'6	p@ssW0rd	p@ssW0	
Replace	sXY	Replace all instances of X with Y	ss$	p@ssW0rd	p@$$W0rd	
Purge	@X	Purge all instances of X	@s	p@ssW0rd	p@W0rd	+
Duplicate first N	zN	Duplicates first character N times	z2	p@ssW0rd	ppp@ssW0rd	
Duplicate last N	ZN	Duplicates last character N times	Z2	p@ssW0rd	p@ssW0rddd	
Duplicate all	q	Duplicate every character	q	p@ssW0rd	pp@@sssssWW00rrdd	
Extract memory	XNMI	Insert substring of length M starting from position N of word saved to memory at position I	lMX428	p@ssW0rd	p@ssw0rdw0	
Append memory	4	Append the word saved to memory to current word	uMl4	p@ssW0rd	p@ssw0rdP@SSW0RD	
Prepend memory	6	Prepend the word saved to memory to current word	rMr6	p@ssW0rd	dr0Wss@pp@ssW0rd	
Memorize	M	Memorize current word	lMuX084	p@ssW0rd	P@SSp@ssw0rdW0RD	

With reference to the syntax for hashcat rules above, it is possible to craft the two rules discussed earlier. Appending `123` onto the end of a password is achieved by using the `$X` function where `X` is the character that is to be appended onto the end of the word. Substituting the character `a` with the `@` symbol is achieved by using the `sXY` function where `X` is the character to be replaced with the character inserted into `Y`. The two rules would look as follows:

- Append `123` - `$1 $2 $3`
- Substitute `a` with `@` - `sa@`

An important function defined in the hashcat syntax for writing rules is the `:` function. This simply attempts to guess a word without performing any modification on the word. Including this function in a set of rules ensures that the wordlist is guessed as is. Following the creation of these rules, a rule file saved in this example as `2_custom_rules` should contain the following text:

```
$1 $2 $3
sa@
:
```

It is possible to re-run the original attack with the addition of these simple rules. The command to do this is similar to the original command, except for the addition of the `--rules` flag. The command to execute the attack with the addition of rules is as follows:

```
./hashcat-cli64.bin -m 0 bfield.hash phpbb.txt -o wordlist_with2rules.txt
    --rules 2_custom_rules
```

The results of this attack are shown below:

```
Initializing hashcat v0.49 with 8 threads and 32mb segment-size...

Added hashes from file ../UniPresentations/bfield.hash: 548686 (1 salts)
Added rules from file 2_custom_rules: 3

NOTE: press enter for status-screen


Input.Mode: Dict (../Dictionaries/GeneralLists/phpbb.txt)
Index.....: 1/1 (segment), 184389 (words), 1574395 (bytes)
Recovered.: 25330/548686 hashes, 0/1 salts
Speed/sec.: 1.14M plains, 381.32k words
Progress..: 184389/184389 (100.00%)
Running...: --:--:--:--
Estimated.: --:--:--:--

Started: Fri Jul 17 12:41:52 2015
Stopped: Fri Jul 17 12:41:53 2015
```

In the interests of experimenting with rules empirically, hashcat can record the effectiveness of each rule by writing it to a file when it successfully cracks a password. The subsequent rule list can be analyzed to discover the effectiveness of each successful rule. To record this information, two additional flags should be passed to hashcat, the `--debug-mode flag` and the `--debug-file` flag. The command to

execute this attack is as follows:

```
./hashcat-cli64.bin -m 0 bfield.hash phpbb.txt -o wordlist_with2rules.txt
    --rules 2_custom_rules --debug-mode=1 --debug-file=successful_rules.txt
```

After running this attack and sorting the resulting `successful_rules.txt` file, the following results are output:

Rule	Number of Passwords Cracked
:	22269
$1 $2 $3	3026
sa@	35

As might be expected, the wordlist itself unaltered performed the best. However, with the addition of two simple rules, the total number of passwords cracked has increased by nearly 3000 guessed passwords. Interestingly, appending `123` onto the end of each password was far more effective than a simple character substitution appeared to be. The test can now be re-run with the addition of some more complicated rules. Some additional substitutions such as swapping `e` with `3`, and `s` with `$` have now been added. These substitutions yield passwords like `p@$$w0rd` from the word `password`. The attack yields the following results after adding these rules to our rule file and ordering the results by the most successful rule:

Rule	Number of Passwords Cracked
:	22245
$1 $2 $3	2978
sa@ se3	360
sa@ se3 ssS so0	252
se3	232
sa@	13
sa@ se3 ss$	7

There is now a marked improvement in the number of passwords that the substitution based rules crack after the addition of some more complex substitution based rules. Interestingly, the `:` rule has cracked fewer passwords. This is because the order in which hashcat applies the rules from the rule file to the wordlist influences the success of each rule. In this way, the order in which hashcat processes the rule file influences the outcome slightly. Despite this, it is clear that based on the results of the attacks, appending a sequence of numbers seems to be an effective strategy in cracking passwords. By performing a dual character substitution, in swapping `a` with `@` and `e` with `3`, a total of 360 passwords were cracked. This is a considerable improvement on a single character substitution that was tested in the previous test. It seems however, that based on some simple tests and experimentation, that appending characters to the end of a password is a more effective strategy to crack passwords with than character substitutions are. It is possible to test this hypothesis by adding a series of rules that simply append a number to the end of a password. After adding a single digit to the end of each password in the wordlist, the results are as follows:

Rule	Number of Passwords Cracked
:	21150
$1	5999
$1 $2 $3	2918
$2	2062
$3	1133
$7	1001
$5	893
$4	873
$0	858
$9	748
$8	731
$6	712
se3	551
sa@ se3 ss$ so0	264
sa@	32
sa@ se3	5

With the simple addition of the 16 above rules, there was a significant increase in the total number of passwords cracked. The result of the final rule based attack has increased the total number of hashes cracked from 22542 to 40020. This equates to over 7% of the battlefield hashset.

```
Input.Mode: Dict (/media/mm/1971c69e-f676-45fb-bfce-f354ba961919/PasswordResearch/Dictionaries/phpbb.txt)
Index.....: 1/1 (segment), 184389 (words), 1574395 (bytes)
Recovered.: 40020/548686 hashes, 0/1 salts
Speed/sec.: 2.02M plains, 126.34k words
Progress..: 184389/184389 (100.00%)
Running...: 00:00:00:02
Estimated.: --:--:--:--

Started: Mon Jul 20 15:25:04 2015
Stopped: Mon Jul 20 15:25:06 2015
```

Developing good rulesets is an important part of password cracking and greatly increases the efficiency of a wordlist. The experimentation conducted thus far has been an example of how to begin developing rules that crack passwords. The reality is that good rules become increasingly complex and less effective as a ruleset grows in size. In many cases, the password that a system administrator uses will be governed by a more complex password policy than typical passwords. Sometimes, an unusual sequence of manipulations may be required to be performed on a root word in order to generate a system administrator's password, and the rule that produces this may be effective just once. In this instance, solely basing the value of a rule on the number of passwords that it cracks may not wholly represent it's usefulness. However, as a start and especially when working with modest hardware, it is important that a ruleset is targeted, tuned and efficient and determining this from the number of passwords that a rule cracks is a reasonable start.

Writing Good Rules

A good ruleset is the mainstay of a password crackers toolset, and crackers typically take great pride in curating these lists and generating unique, complex rules that crack highly complex passwords. This document intends to serve as a basic introduction to rule based attacks on passwords. A good start to creating a more complex ruleset is to consider how people think when they choose passwords and attack this psychology. It is then possible to experiment with developing more complex rules by creating rules that append and prepend the year and month, insert some common names and keyboard patterns and alter the case of a word in interesting ways, for example. Empirically testing these ideas can then be achieved by observing both the quantity and kind of password that each rule cracks. While appending `123` may be effective at cracking a large number of simple user passwords, an administrator password might be successfully cracked by capitalising the first and seventh letter, inserting an exclamation mark in between the eighth and ninth character, overwriting all occurrences of the letter `a` with `@` and appending `qwerty` to the end of the password. It is also worth noting that a ruleset that is developed and targeted toward cracking battlefield passwords might not fare as well against a large enterprise password set. With different environments, password policies and password postures across organisations, expect results to differ significantly between tests. MWR's ruleset has been crafted over several years and contains over a million rules. The results of our ruleset combined with phpbb.txt against battlefield are shown below:

```
Session.Name...: oclHashcat
Status.........: Exhausted
Rules.Type.....: File (mattrules2.rule)
Input.Mode.....: File (../Dictionaries/phpbb.txt)
Hash.Target....: File (bfield.hash)
Hash.Type......: MD5
Time.Started...: Mon Jul 20 15:41:38 2015 (2 mins, 45 secs)
Time.Estimated.: 0 secs
Speed.GPU.#1...:  1151.2 MH/s
Speed.GPU.#2...:        0 H/s
Speed.GPU.#*...:  1151.2 MH/s
Recovered......: 243208/423623 (57.41%) Digests, 0/1 (0.00%) Salts
Progress.......: 307553107662/307553107662 (100.00%)
Rejected.......: 0/307553107662 (0.00%)
```

Hashcat ships with a number of good rulesets that can be found in the /rules directory in hashcat to get started with, but these are just a starting point! Experiment and build up your rulesets. Happy cracking!

[1] http://hashcat.net

[2] https://hashcat.net/wiki/doku.php?id=hashcat_utils

[3] https://thesprawl.org/projects/pack/

[4] https://www.reddit.com/r/crypto/comments/yuqyi/john_the_ripper_vs_oclhashcatlite/

[5] http://www.adeptus-mechanicus.com/codex/hashpass/hashpass.php

[6] https://wiki.skullsecurity.org/index.php?title=Passwords

[7] https://hashcat.net/wiki/doku.php?id=rule_based_attack