Hakin9

PRACTICAL PROTECTION

IT SECURITY MAGAZINE

OFFENSIVE COMPUTER SECURITY

KALI, BASH, NETCAT, WIRESHARK ...

PASSWORD CRACKING – BEYOND BRUTE-FORCE

MAKING THE CALL: SIX KEYS TO AN EFFECTIVE BYOD TECH POLICY

PORT SCANNING | FINGERPRINTING | ARPSPOOFING | MITM | BUFFER OVERFLOW EXPLOITATION | EXPLOIT DB

Vol.10 No.02
Issue 02/2015(77) ISSN: 1733-7186

Password Cracking – Beyond Brute-Force

by Immanuel Willi

Most password mechanisms work by comparing a password against a stored reference value. It is insecure to store the whole password, so one-way functions are used to create hash values from the passwords. A one-way function ensures that a password is mapped to a seemingly random (hash) value, but the (hash) value cannot be mapped back to a clear-text password.

If the plain-text string is only minimally changed, ideally a completely different hash value is produced. Since the password cannot be derived from the hash, all potential passwords need to be "hashed" with the same hash function to determine the validity based on a comparison with the known hash value of the valid password. The process of breaking hashes by guessing the original password is called „hash cracking".

Table 1. Sample MD5 hashes

String	MD5 hash
Oneconsult	00f2788f99a47aa6e9cb8afe66b5f033
oneconsult	1963754b8c04afa68f520cf214aebdeb
123456	e10adc3949ba59abbe56e057f20f883e

Instead of storing a password on a system to enable the authentication of a user, the hash value of the password is stored to ensure that other users cannot directly read out the password. In contrast to operating systems which store user passwords locally (in Windows the SAM file, in Linux „/etc/shadow" and in Mac OS X „/var/db/dslocal/nodes/Default/users/<user>.plist"), the password hashes of online shoppers are stored in databases on the webserver. Another typical place where password hashes may be found is local scripts or the Windows registry, which make passwords available for authentication in hard-coded form.

For a penetration tester, a found hash is an opportunity to expand the penetration test to surrounding systems, if the password can be reconstructed from the hash. The same is true for an attacker. Reading out the hashed passwords of all registered users of a database (e.g. by an SQL injection) may be attractive for criminals, as such data may be sold on respective platforms. Not only penetration testers and hackers are interested in breaking password hashes. By now, an active community of hash crackers has come to life with known groups who regularly battle against each other in competitions. Participating groups receive lists of password hashes which they need to crack within a given timeframe. It is from these competitions and from the community that new ideas and tools are created with which hash cracking may be optimized.

The most obvious method to find the clear text of a hash value is the brute-force attack. There are no pre-calculated values or tables used, but instead the attacker calculates all possible combinations of numbers, letters and special characters until a hash value is found which corresponds to the original hash value. However, brute-force attacks on hash values cannot be carried out within a reasonable timeframe given a certain length and complexity of the clear text, as trying out different potential combinations (with traditional means) increases exponentially with the password length. An alternative to systemically searching through the whole space of conceivable passwords is to perform precomputation of a list of likely password candidates and generate a list of known password hashes this way. For a given hash value, it is then possible to simply search this list to find the associated password. Searching a list is much faster than brute forcing all conceivable passwords.

This precomputed hash list attack does not calculate the values at runtime like the brute-force method does. Since the computation of hash values is carried out before the actual attack on the hash values, it does not need to be done more than once. Therefore, it is possible to download such hash lists from the internet or even search online for matching passwords given a known hash value. This process requires hardly any computing time of the CPU, however, the dictionaries may ─ depending on their size ─ consume a lot of memory and the precomputation is as slow as a regular brute-force attack.

A brute-force attack will take a lot of time, but will hardly use any physical memory for the attack. The perfect attack would thus require an infinite amount of time. The opposite is true for a hash list attack, which could theoretically list all possible passwords in a pre-calculated table. This alternative would use an infinite amount of memory, but relatively little computing time. A combination of time and memory would thus be ideal. The attack should be carried out using predefined tables to reduce computing time, but the tables should be limited to a manageable size by means of the computations.

This idea was developed by Martin Hellman in 1980 in a time-memory tradeoff (TMTO) algorithm. With a TMTO, predefined data for cryptanalysis is temporarily stored in memory. In an extended version of the algorithm, developed by Ronald Rivest in 1982, the look-up in memory was massively reduced, which led to an increase in speed. In 2003, Philippe Öchslin created the concept of the rainbow tables based on the TMTO, which again halved the number of necessary computations. However, using rainbow tables can cause problems. On the one hand, tables need to be calculated for every hash function and for each individual character set; on the other, since the emergence of rainbow tables, hash values are usually salted. Salting denotes the technique of mapping passwords to hash values depending on a random salt value in order to prevent precomputation. For example, the salt value can be appended to the clear text before it is hashed. Rainbow tables would need to be much larger to also include all possible combinations of additional characters next to the actual clear text. Thus, by salting hash values, the effective use of rainbow tables is impeded.

Table 2. Example of an unsalted hash

Hash function(string)	SHA1 hash
sha1(oneconsult)	9e422594d374752732eeeac9529eb14dec818e57

Table 3. Example of a salted hash

Hash function(salt+string)	SHA1 hash
sha1(123456+oneconsult)	a348809498f93e132bd211b8ec4838be416e0abf

Current Methods and Approaches

Optimizing Cryptanalysis on Hardware

Computations for cryptanalysis have long been carried out only on the CPU. Modern Graphical Processing Units (GPUs), the processors on graphic cards, are optimized to graphically display sophisticated computer games and 3D applications in real time. In games, millions of polygons need to be computed in parallel for the gaming world to appear as realistic as possible. The development of graphic cards has been optimized in the past 10 years for the parallelization of small computations. Manufacturers outdo each other with each new generation of graphic cards and their technical performance. The parallel computation of "simple" calculations corresponds exactly to the requirements for the computation of many single hash values in cryptanalysis. Tools like John The Ripper[1] or oclHashcat[2] use these capabilities to compute hashes at a tremendous speed on the GPU instead of the CPU.

Table 4. Comparison of CPU to GPU

	CPU	GPU
Test hardware:	Intel Core i7-4770 CPU @ 3.40GHz	ATI Radeon R9 290x, Hawaii, 3072MB, 1040Mhz, 44MCU
Software:	hashcat-0.49 (--benchmark -m 0)	oclHashcat-1.36 (--benchmark -m 0)
Hash function:	MD5	MD5
Speed:	MH/s: 99.08	MH/s: 12035.6

Cryptanalysis on the GPU is 121.5 times faster than on the CPU. The CPU in the test setup computes 99 million MD5 hashes per second, the GPU 12 billion.

1 John the Ripper with jumbo patch: http://www.openwall.com/john/
2 OclHashcat: http://hashcat.net/oclhashcat/

An alternative to buying hardware is to use computing power in the cloud. Various vendors, such as Amazon with the EC2 Cloud, offer GPU instances for hire. The used hardware may be scaled up by adding additional GPUs. Although prices look affordable at first sight, it is worth considering whether the hiring costs should not rather be invested in hardware.

Optimizing Cryptanalytic Approaches

Limitations of Brute-Forcing

Without precomputed components, cryptanalysis has to rely on traditional methods. Attacking a hash by brute force is only useful to a certain degree. One of the reasons is that as password length increases, possible combinations grow exponentially. Let's take a look at OclHashcat to illustrate our point:

Standard character set of oclHashcat:

```
?l = abcdefghijklmnopqrstuvwxyz
?u = ABCDEFGHIJKLMNOPQRSTUVWXYZ
?d = 0123456789
?s = !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
?a = ?l?u?d?s
```

Thus, the mask (`?a?a?a?a?a?a?a`) [7] corresponds to a seven-character password, which is made up of lowercase and uppercase characters, digits and special characters.

This combination yields $(26+26+10+32)^7 = 94^7$ different password options, which corresponds to $6.484775942 \times 10^{13}$ possible passwords (64 trillion, 847 billion, 759 million, and 420 thousand). As shown below, this character set (for a MD5 hash function) can be traversed on the GPU within hours:

```
Input.Mode.....: Mask (?a?a?a?a?a?a?a) [7]
Hash.Target....: 1963754b8c04afa68f520cf214aebdeb
Hash.Type......: MD5
Time.Estimated.: 2 hours, 7 mins
```

As possible combinations increase exponentially with each additional character, a brute-force attack on an eight-character password with lowercase and uppercase characters, digits and special characters takes several days:

```
Input.Mode.....: Mask (?a?a?a?a?a?a?a?a) [8]
Hash.Target....: 1963754b8c04afa68f520cf214aebdeb
Hash.Type......: MD5
Time.Estimated.: 7 days, 5 hours
```

The attack on a nine-character password already lasts longer than a year:

```
Input.Mode.....: Mask (?a?a?a?a?a?a?a?a?a) [9]
Hash.Target....: 1963754b8c04afa68f520cf214aebdeb
Hash.Type......: MD5
Time.Estimated.: 1 year, 353 days
```

The computing power of the hardware may be linearly increased by adding further graphic cards, however, if longer and more complex passwords are used, this increase cannot keep up with the exponential growth of the number of potential character combinations.

A further challenge for efficient brute-forcing concerns the fact that modern hash functions are designed to make the calculation of the function less performant. The user of a Linux/BSD operating system will not notice any delay when the Unix crypt SHA-256/512 is applied. However, the performance of the hash function which computes the hash from the entered password is artificially decreased as may be gathered

from an excerpt of the SHA-256/512 crypt specification by Ulrich Drepper[3]: "…the SHA-based algorithm contains a loop which can be run an arbitrary number of times. The more rounds are performed the higher the CPU requirements are. This is a safety mechanism which might help countering brute-force attacks in the face of increasing computing power."

Table 5. Comparison of MD5 with SHA-512 crypt

	MD5	SHA-512 crypt
Test hardware:	ATI Radeon R9 290x, Hawaii, 3072MB, 1040Mhz, 44MCU	
Software:	oclHashcat-1.36 (--benchmark -m 0)	oclHashcat-1.36 (--benchmark -m 1800)
Speed:	12035.6 MH/s	71437 H/s

Cryptanalysis of an MD5 hash may be carried out with a speed of 12 billion hashes per second. In contrast, only about 70.000 SHA-512 crypt hashes may be analyzed per second. The increasing use of complex passwords and the adoption of less performant hash functions underline that brute-forcing is only appropriate in the context of performant hash functions, such as MD5, or very limited character sets. For other scenarios, cryptanalysis needs to be optimized with other techniques, some of which will be briefly presented in the next section.

Attack Vector Psychology

Many users are not very creative when it comes to choosing a password. One way to take advantage of this fact is to use **password lists** with genuine user-generated passwords. Based on different hacking attacks, several lists are in circulation, such as the rockyou.txt password list with over 14 million different user-generated passwords. **Wordlists** are another option to break trivial passwords. From the Oxford Englisch Dictionary to the German Duden, various wordlists in text file format are made available online. If the origin of a password hash is known, this context may be taken into account when searching for or creating wordlists. One could also make use of lexicons of different domains such as medicine, aviation, music, or employ gazetteers (containing geographical landmarks) or lists with film titles. These lists may target the user directly, i.e. his or her professional environment or personal interests.

Crawlers are also interesting in that they spider whole websites and create an index, i.e. a wordlist from the text of such a website. Passwords of many users follow similar patterns. Many passwords are between 6 and 8 characters long and mostly consist of lower-case characters, with the option of the first letter being upper-case. Digits and special characters are mostly appended to passwords.

If a hash value originates from a company environment where password complexity policies (such as 8 characters, lower and upper case, digit or special character) are implemented, the search space is restricted, as all password candidates which do not correspond to the requirements do not have to be tried out. Too restrictive and complex password quality requirements may thus be paradoxically counterproductive with respect to password security.

The use of **masks** in in oclHashcat allows specifying the length and structure of passwords to be searched for. A possible mask for a password with the above mentioned requirements would be "?u?l?l?l?l?l?d". The password "Consult1" would, for example, be a fitting password.

In addition, oclHashcat allows you to develop complex rules based upon which wordlists may be built or be manipulated. A rule could, for example, traverse all words of a wordlist and replace all "a"s by "@", all "o"s by "0" to generate "P@ssw0rd123" out of "Password123". The rules may be increased in complexity as required. Predefined rulesets of various cryptanalysis groups are available online or already supplied by oclHashcat. However, the most effective rulesets are treated as "secrets of success" and therefore not disclosed to the public. Password lists and wordlists may be extended by masks and rules. This mode is called **hybrid attack**. The mask or rule may thus be prepended or appended to the (pass)word.

Example: Say you search for a hash value of the unknown password "Susanna1984" and the rockyou password list does not yield a match, as precisely this password is not covered by the list. However, in the rockyou.txt password list, the following passwords starting with "Susanna" are included:

3 Ulrich Drepper SHA-256/512-Crypt specification, reference implementation, and test vectors http://www.akkadia.org/drepper/SHA-crypt.txt

Susannah	Susannah58	Susannah123	Susanna99
Susanna91 Susanna1	Susanna79 Susanna	Susanna33	Susanna27

If the mask ?d?d?d (digit, digit, digit) is appended to the password list, passwords in the rockyou list are extended to include:

Susannah?d?d?d	Susannah58?d?d?	Susannah123?d?d?d	Susanna99?d?d?d
Susanna91?d?d?d Susanna1?d?d?d	Susanna79?d?d?d Susanna?d?d?d	Susanna33?d?d?d	Susanna27?d?d?d

In the entry „Susanna1?d?d?d“ all digits from Susanna1“000“ to Susanna1“999“ are tried out, which will lead to a match with Susanna1“984“.

Rules allow for very complex manipulations of words in word- and password lists. A rule could define that all „o“s and all „s“s be replaced by „0” and „$“ respectively to generate the string „Pa$$w0rd“ out of „Password“. Another function returns for instance the word from the wordlist spelled backwards such that „Password” returns the string „drowssaP“. Many more rules are applicable and may be combined as needed. John the Ripper, as well as hashcat, supports the use of rules.

The **combinator attack** puts each word of a password list or a wordlist together with each word from one or more lists. The combinatory attack may also be applied two or more times to the same list. This attack can be combined with the other attack methods mentioned above as desired. This attack can be useful to crack word doubling or pass phrases. A very performant enhancement to the word combination approach was published by Jens Steube, the key developer of hashcat, called **prince attack**. The prince attack combines all input words in all possible combinations, whereby the number of elements to be generated may be defined.

Attacks Based on Statistics

If an entire list of hashes needs to be analyzed, as, for example, when doing an IT security audit where the strength of passwords chosen by users are examined, it is worth analyzing already broken hashes to efficiently attack the remaining hashes. The advantage of this method is to discover patterns which users in a specific environment employ when choosing a password.

When breaking the hashes in the „linkedin“ hash list, it becomes clear, after the discovery of the first half of the passwords, that many users employ a variation of the word „linkedin“ as a password. With this in mind, one can specifically search for mutations and combinations of the string “linkedin”, which results in the discovery of more complex passwords.[4] PACK, short for „**Password Analysis and Cracking Kit**“[5], provides extensive analysis capabilities for this purpose. The toolkit examines user passwords and produces custom masks for further attacks. Thus, with PACK it is possible, amongst others, to benchmark the systems and hash function in order to generate all masks which may be traversed in a freely defined timeframe.

4 *See Yiannis Chrysanthou, “I have the HASHCAT so I make the rules”http://hashcat.net/events/p14-vegas/I%20have%20the%20%23cat%20i%20make%20the%20rules_YC.pdf*

5 https://thesprawl.org/projects/pack/

Sample analysis of the rockyou.txt password list:

```
                        _
         StatsGen #.#.#   | |
          _ __   __ _  ___| | _
         | ,_ \ / _` |/ __| |/ /
         | |_) | (_| | (__|   <
         | .__/ \__,_|\___|_|\_\
         | |
         |_| iphelix@thesprawl.org

    [*] Analyzing passwords in [rockyou.txt]
    [+] Analyzing 100% (14344390/14344390) of passwords
        NOTE: Statistics below is relative to the number of analyzed passwords, not total number
of passwords

    [*] Length:
    [+]                          8: 20% (2966037)
    [+]                          7: 17% (2506271)
    [+]                          9: 15% (2191039)
    [+]                         10: 14% (2013695)
    [+]                          6: 13% (1947798)
    ...

    [*] Character-set:
    [+]              loweralphanum: 42% (6074867)
    [+]                 loweralpha: 25% (3726129)
    [+]                    numeric: 16% (2346744)
    [+]       loweralphaspecialnum: 02% (426353)
    [+]              upperalphanum: 02% (407431)
    ...

    [*] Password complexity:
    [+]                      digit: min(0) max(255)
    [+]                      lower: min(0) max(255)
    [+]                      upper: min(0) max(187)
    [+]                    special: min(0) max(255)

    [*] Simple Masks:
    [+]                stringdigit: 37% (5339556)
    [+]                     string: 28% (4115314)
    [+]                      digit: 16% (2346744)
    [+]                digitstring: 04% (663951)
    [+]                  othermask: 04% (576324)
    ...

    [*] Advanced Masks:
    [+]           ?l?l?l?l?l?l?l?l: 04% (687991)
    [+]               ?l?l?l?l?l?l: 04% (601152)
    [+]             ?l?l?l?l?l?l?l: 04% (585013)
    [+]         ?l?l?l?l?l?l?l?l?l: 03% (516830)
    [+]             ?d?d?d?d?d?d?d: 03% (487429)
    ...
```

The mathematical model of **Markov chains** may be used to generate word lists based on the analysis of existing texts. Instead of trying out all possible combinations of passwords, a statistical model decides which characters based on the analysis of the wordlist appear frequently one after the other.

Conclusion

In this article, we have discussed the basics of hash cracking and the various ways to optimize this process. Getting an overview of the diverse methods and tools is a starting point, but it is much more difficult to combine them efficiently and in a helpful way. The fact that complex and supposedly secure passwords like „n3xtb1gth1ng“, „m27bufford“ or „Oscar+emmy2“ may actually be broken, is impressively demonstrated by the results of regularly held competitions. Considering these facts may shed a new light on how we choose our own passwords, the consequences of which we will leave up to the reader.

About the Author

The author Immanuel Willi works as a Security Consultant at Oneconsult AG, which specializes in penetration tests, ISO 27001 security audits and IT forensics. He has been exploring and researching the topic of password cracking for many years.