Cloud computing broke password hashing

Rais Mense (2510240)

May 6, 2013

Abstract

The advent of cheap cloud computing resources has given the general public access to computational resources that were limited to a just select few individuals no more than a few years ago. Now that this power is available to anyone who wants it we should take a look at how this affects the security of commonly used password hashing algorithms. A password hash is only useful if it takes longer to crack than the time the password is valid, and this balance is now shifting for a number of hashing algorithms. We will not only look at specific algorithms but also draw some conclusion about why some do hold up while other don't and what the fundamental properties are that enable this.

1 Introduction

As long as people have stored personal information on their computers there has been a need for passwords to protect them. The conventional way of securely storing passwords on any computer system always was, and still is through password hashing. A cryptographic algorithm is applied to the password the user enters, resulting in a hash that cannot be reversed back to its original password. Almost as soon as passwords were being stored in hashes people have tried to get the original password back from the hash for a number of reasons, both good and bad. The crucial factor for encryption has always been that the time required to break the encryption should be longer than the time that the encrypted data is useful. It then logically follows that a password hashing algorithm is only as good as the computing power required to get the original password back. For the purposes of this paper we will consider a hashing algorithm broken when its hashes can be cracked within a time space where it is likely to still be useful.

1.1 Clouds

With the rise of cloud computing it is becoming increasingly easy for anyone to have access to very large amounts of computing power on demand for a relatively low cost. For just over $2 an hour anyone with a valid credit card can go to Amazon and make use of one of their very powerful GPU accelerated machines delivering a peak performance of about a Teraflop for double precision operations. This is a scale of computing power that was previously impossible to get access to without spending hundreds if not thousands on the hardware.

Previously only large institutions had access to these kinds of resources and the people who could run a password hash cracking job at their leisure on such systems were few and far between. Now that this power is available to most people, we need to re-evaluate the usefulness of some commonly used hashing algorithms to see if they should still be used or not.

1.2 Hashing algorithms

Not all types of password hashing are equally susceptible to increasing computing power. In this paper we will focus on a set of four commonly used hashing algorithms.

The md5 hashing algorithm has long been used for both password hashing and data integrity checks. In recent years the md5 algorithm has suffered several setbacks and is no longer considered appropriate for password hashing[1][2], however given its long history of usage and that in spite of these recent findings it is still being commonly used we will include it in our examination.

The sha-1 algorithm is commonly considered a suitable replacement for applications where md5 is currently being used as the password hashing algorithm. In practice this turns out to not be the case as sha-1 has itself been broken since 2005.[2] Again, we will still include it for its common usage.

The NTLM hashing algorithm was developed by Microsoft as a replacement for their LM hashing algorithm. Though the NTLM algorithm has successfully resolved many of its predecessors shortcomings it has still been viewed in some negative light, not over cryptographic algorithm itself but rather for it's implementation. It is currently being used as the default hashing algorithm for all modern Windows operating systems and will thus be included in our examination for its wide spread usage.

The final hashing algorithm will be bcrypt which is an implementation for password hashing based on the general purpose blowfish algorithm which was written to serve as an alternative to the DES encryption algorithm.[3] It is currently commonly used as the password hashing algorithm for several Unix and Linux operating systems. The bcrypt implementation is generally considered to be one of the few "best practice" methods for password hashing [1] so it should be included in our examination.

2 Attacking the hashes

2.1 Method of attack

There are many methods to attack hashed passwords. Some common methods include generating hashes from a list of passwords or a dictionary, using precomputed rainbow tables to look up the hash and its corresponding value and a brute force attack that iterates through each possible combination of a set of characters. We will be focusing on the brute force attack as it is at the root of the other two attacks. The dictionary attack is essentially pruning less likely combinations where the rainbow table does a brute force attack in advance and stores the results in a look-up table. We have discarded the rainbow table method here because it will not work against salted passwords and bcrypt hashes are always salted so the comparison to the other hashing algorithms would not result in a fair comparison. The dictionary attack may also be used to evaluate the hashing functions in a fair comparison for computation time however we will not be using it as to avoid the possibility of introducing any bias by the words included in or excluded from the dictionary. As our example for modern hash cracking tool we will use an application called "oclHashcat" which was designed to run on highly parallel processors such as GPUs. It it also capable of attacking all four of the hash types we will be looking at thus eliminating any variance introduced by the implementation as much as possible. This program will also allow for the cracking of salted md5 and sha-1 hashes. The figures for the performance of oclHashcat for each type of hash are known and have been published on their website. We will use these figures as a starting point to evaluate how long it would take to crack a password.

2.2 The figures

The oclHashcat benchmarks include a benchmarks for a PC with an NVIDIA GTX 560 Ti GPU, which is a relatively close match in performance for the cards used in the Amazon EC2 GPU accelerated instances. As an Amazon EC2 instance has two of these GPU based accelerators we can simply double the figures for the oclHashcat benchmark, assuming that the performance scales roughly linearly. It is possible that the real world scaling does not quite match this performance but this would be a matter of the implementation or a consequence of the two cards sharing some of their other resources like an IO bus. This is not a concern as the orders of magnitude difference between the hash types does not change. We could also simply run the application on two instances if the scaling should prove to be dramatically less than linear, albeit at the expense of doubling our cost of course. The following table lists the measured and expected number of cryptographic operations or attempted password cracking tries per second for each hash type. The expected EC2 numbers are rounded down.

Hash type	oclHashcat benchmark	Expected EC2 performance
MD5	1.345×10^9	2.6×10^9
SHA-1	4.33×10^8	8×10^8
NTLM	1.641×10^9	3.2×10^9
bcrypt	604	1200

We can see that the number of attempts we'll be able to make on bcrypt is significantly lower than for the other hashing algorithms. In the following sections we will see if this translates to a real-world benefit or if in spite of increased computational power required the attack is still going to be short enough to be useful.

2.3 The math

In this section we will present the calculations we will use to determine how long it should take to crack a hash. The search space for a password will be defined as all possible combinations of characters in the character set for the full length of the password. To calculate the search space we just do $searchspace = c^l$ with c as the number of characters in the character set and l the length of the password.

The following table lists the five passwords we will be looking at for our examples, each with different properties.

Password	Character set	Search space	Password type
password	[a-z]	26^8	Simple
PassWord	[a-zA-Z]	52^8	Mixed case
P4ssW0rd	[a-zA-Z0-9]	62^8	Alpha numeric
P4$$W0rd	[:all:]	95^8	Full key space
passwordpassword	[a-z]	26^{16}	Long password

It is interesting to note that the common concept of adding numbers and special characters is not as effective for creating a large search space as having a longer but simple passphrase. Even a simple password of 16 characters will give a larger search space than a complex password of eight.

If we calculate the search spaces for all the characters in the set from length 1 to the length of the password and sum those we will have the cumulative search space for that password. When we take half of the search space and add to that the cumulative search space for password length-1 we will have a reasonable approximation to the number of tries it will on average take to crack a password of that length.

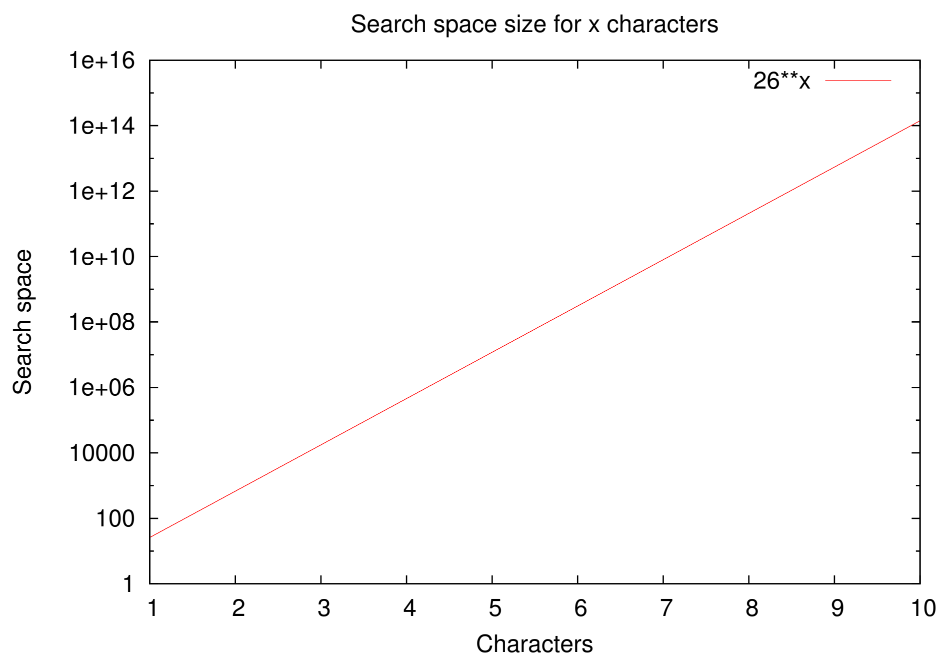
We can calculate the maximum time in seconds it should take to crack each hash using the function:

$$S_{max} = \sum_{i=1}^{l} \frac{c^i}{a}$$

Where c is the size of the character set, l is the length of the password and a is the number of attempts we make to crack the hash every second. However for the sake of simplicity we will be using the function

$$S_{max} = \frac{c^l}{a}$$

which discards the computational time for all preceding characters. As every following character takes an order of magnitude longer than the preceding one it will still give us a reasonable indication of how long it will take. The preceding characters may take a long time, but they will not add significantly to the total time anymore. The following graph illustrates that the search space increases exponentially with every extra character. Note that the Y-axis is logarithmic.



Now we simply divide by two in order to get the average amount of seconds it will take to crack a hash: $S_{avg} = \frac{S_{max}}{2}$. Using these functions we can construct the following tables. All times in seconds have been rounded to the nearest second.

"password"

Algorithm	S_{max}	S_{avg}	Human readable time
MD5	80	40	40 seconds
SHA-1	261	130	2 minutes
NTLM	65	33	33 seconds
bcrypt	174022554	87011277	2.7 years

"PassWord"

Algorithm	S_{max}	S_{avg}	Human readable time
MD5	20561	10281	Just under 3 hours
SHA-1	66825	33412	Just over 9 hours
NTLM	16706	8353	Just under 2 and a half hour
bcrypt	4.4550×10^{10}	2.2275×10^{10}	About 706 years

"P4ssW0rd"

Algorithm	S_{max}	S_{avg}	Human readable time
MD5	83977	41988	11 and a half hour
SHA-1	272925	136463	About a day and a half
NTLM	68231	34116	About 9 and a half hours
bcrypt	1.8195×10^{11}	9.0975×10^{10}	About 44 centuries

"P4$$W0rd"

Algorithm	S_{max}	S_{avg}	Human readable time
MD5	2551617	1275809	Just under 15 days
SHA-1	8292755	4146378	Just over 19 days
NTLM	2073189	1036594	Almost Almost 12 days
bcrypt	5.5285×10^{12}	2.7643×10^{12}	About 88 centuries

"passwordpassword"

Algorithm	S_{max}	S_{avg}	Human readable time
MD5	1.6772×10^{13}	8.3863×10^{12}	Almost 266 centuries
SHA-1	5.4511×10^{13}	2.7255×10^{13}	Just over 864 centuries
NTLM	1.3628×10^{13}	6.8139×10^{12}	About 216 centuries
bcrypt	3.6341×10^{19}	1.8170×10^{19}	About 576 million centuries

The differences between cracking times for the different algorithms are quite extensive and in many cases several orders of magnitude apart. In the next section we will try to draw some general conclusions about what this means for the hashing algorithms and how they should be used.

3 Conclusion

From the results of our calculations we can see that cracking even fairly complex passwords can be done in a reasonable amount of time. We can also see that longer passwords are extremely effective at increasing the cracking time. Therefore the fact that both md5 and sha-1 can be implemented using salts gives them a significant advantage over NTLM which is generally only used in the Windows operating system under an implementation where salting is not used. Though the NTLMv2 standard does employ salting when sending hashes over the network for remote login, the hashes being stored are still not salted. Bcrypt requires a salt to be used at all times which is part of the reason why it is so difficult to crack. However that is not the only reason bcrypt preforms so well. The main reason why it takes so long is the configurable cost variable bcrypt uses. This is set to a cost of ten by default in most implementations, including the one on which our figures are based. This gives a reasonably secure hash as we can see from our calculations. Even fairly simple passwords cannot be cracked in a realistic amount of time. On the surface it would seem that a good implementation of the md5 or sha-1 algorithm with the use of a salt could easily push them back into the territory of being completely unrealistic to crack in any reasonable time span. Unfortunately there is a fundamental problem associated with this type of hashing algorithm. As computing power keeps growing we would need to have every persons password rehashed every so often in order to implement ever longer salts. Bcrypt can solve this problem as it was designed to have its cost included in the hash so when checking the hash against a password the cost associated with that specific hash is known. When it seems prudent to create stronger passwords hashes a rolling update can be done by simply updating the cost value used to create new hashes. Because the cost with associated with each hash may be different, higher costs can be implemented without the need to have all password hashes updated at once. There is also another common problem with using salts in that they are often stored with the password hash so if the hash is known in most cases the salt is too, making the computation only trivially more expensive than it would be with no salt at all.

Though both md5 and sha-1 are considered cryptographically broken now[2], the principle applies to all hashing algorithms that do not have some way to dynamically configure their cost. Even if they are still strong enough now it is only a matter of time before increased access to computing power will put them within reach for brute force cracking too.

In conclusion, easy access to cloud computing resources *did* break password hashing, and will continue to break ever more hashing algorithms as time goes on. But not every hashing algorithm will eventually fall prey to increased availability of resources. As the resources we are able to put towards cracking their hashes grow they can simply increase the cost of their algorithms accordingly.

4 Future research

In this paper we have only considered the case where we are trying to attack a single hash. However in real life we often see cases where large lists of usernames and password hashes fall into the wrong hands. In such a scenario there is a matter of probability to contend with as many passwords will be short and based on dictionary words. Even if many passwords may take a long time to crack, often a small subset of all passwords being cracked can lead to serious problems. Thus it should be useful to employ some statistical experiment to find out how the use of different hashing algorithms may help to reduce the percentages of how many passwords are likely to be compromised.

References

[1] S. Boonkrong, “Security of passwords,” 2012. http://suanpalm3.kmutnb.ac.th/journal/pdf/vol16/ch17.pdf.

[2] J. B. et al, “All in a day’s work: Password cracking for the rest of us,” 2009. http://www.unimed.sintef.no/upload/IKT/9013/dayswork.pdf.

[3] N. Provos and D. Mazières, “A future-adaptable password scheme,” 1999. http://static.usenix.org/event/usenix99/provos/provos.pdf.