Storing your passwords properly

Security Blanket

High-performance graphics cards and proper storage can help keep your passwords secure. *By Kurt Seifried*

Computer performance has come a long way in the past few years. Moore's law has continued moving forward; right now, you can easily find a high-end CPU with 6 to 12 cores for between a few hundred and a thousand dollars.

But what if you want to throw a couple hundred cores at a problem? You could buy a rack or two of equipment, or you could just spend a few hundred dollars on a video card. In fact, graphics cards are so good at certain types of computation that both NVidia and AMD now make specialized cards (the NVidia Tesla series and the AMD FireStream series) that have a ton of cores, several gigabytes of memory, and extremely fast interconnects.

So, assuming you have a decent graphics card (or are willing to buy one), what can you do with a few hundred cores? The most obvious answer is encryption, which is embarrassingly easy to parallelize and works very well on GPU-based computing systems.

SSL Support

The good news is that enabling SSL on most web servers won't take more than a few percentage points of CPU power. The bad news is if you're building a front-end load balancer capable of providing SSL termination for, say, a few thousand clients, you're probably going to need to buy a specialized SSL acceleration card.

Or, you could use a graphics card to provide more cores to handle key setup and data [1]. However, little software is available to provide support for this approach outside of a master's thesis by Urmas Rosenberg (University of Tartu in Estonia) [2] and some associated code that provides AES-128 block support for OpenSSL on CUDA (NVidia-based) cards.

Unfortunately, this is more theoretical than practical right now, "The general result is left as an exercise to the reader." In general, I think generic CPUs are getting so many cores that the need to throw a few hundred cores at this problem will basically become less important.

Password Cracking

Rainbow tables provide a very practical application of GPU-based encryption computation for something useful. The general idea is that people are terrible at choosing passwords. Passwords like "password" or a pet's name – even with some additional characters – simply do not provide a lot of entropy (randomness). Even if the password is salted, you can still do pre-computation to cut down on the search time when you do want to crack a password. The idea is to do a lot of work upfront so that later on you can quickly retrieve passwords.

Some Numbers with AES-256

If you are using AES-256 to hash and store your passwords, you're doing it wrong (more on this in the next section). But, a lot of people are using AES-256 or SHA-256, and it makes for some impressive numbers. Suppose you want to precompute the hash values for all valid characters on a US-English keyboard (26 letters, 10 numbers, 11 other character keys for a total of 94 characters) up to a password length of eight characters (so, $94^1 + 94^2 + \ldots + 94^7 + 94^8$ possible passwords). Storing the input and the AES-256 value (ignoring indexing requirements) for this would result in about 1,400,000TB of data.

KURT SEIFRIED

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He often wonders how it is that technology works on a large scale but often fails on a small scale.

Although hard drive prices have dropped, they haven't dropped enough to make this possible. But you can cheat by precomputing a chain of values, starting with a value of *1*, for example, hashing that, then hashing that result and repeating until you've done it 200,000 times.

You then store the end value of this chain along with the starting value, which means about 14TB of storage (plus indexing requirements, etc.) – or, less than US$ 1,000 in today's hard drives. When you have a hashed password that you want to crack, you compare it to the stored values, if it matches, you win; if not, you hash the hashed password and look for that value.

In the worst case scenario, you have to search around 200,000 times, but eventually you hash the hashed password and end up with a value for which you have a valid chain. Assuming you have hashed the password and searched 50,000 times until you found a match, you then take the starting value of that chain and hash it 149,999 times to get a value. This value, when hashed, will match the password that you're trying to crack.

Why are MD5, AES-256, and SHA-256 such bad choices for storing your hashed passwords? Because they are very fast algorithms, especially on modern hardware, and especially on modern GPUs. The PostgreSQL project has posted some numbers [3] – on a 1.5GHz Pentium 4, you can do 2,345,086 MD5 hashes per second. On a modern GPU, this increases to hundreds of millions per second. However, you don't actually need to make your own rainbow tables, you can download them (and the software used to create them) from a number of free sources [4] [5]. Two main methods are used to defend against rainbow tables: The first is salting, and the second is using encryption and hash functions, like bcrypt, designed for password storage.

Salting Passwords

Salting passwords primarily defends against situations in which the attacker obtains the encrypted or hashed password (e.g., by stealing the `/etc/shadow` file or downloading the database) to brute force it. The salt ensures that a pre-computed brute-force attack will take longer because each password must be encrypted with all possible salt values first.

Some significant weaknesses are inherent with salting, however. The first is that it won't really help against badly constructed passwords: Attackers can easily brute-force a list of the million most common passwords even if proper salting is used.

The second issue is that most systems don't care about the password, they care whether the encrypted or hashed value of the password matches the system entry. Because hash functions like AES-256 only provide 2^{256} possible unique outputs, collisions are obviously possible. Ultimately, the attacker wants to find a data string that will encrypt or hash to the same value as the stored one. And, by brute-forcing chains of possible values, chances are the attacker can find a value that works.

Password Storage

The ugly truth is that most encryption and hash functions are designed to provide encryption and hashing, not secure password storage. However, the bcrypt tool [6] is designed just for password storage. Basically, it uses the Blowfish encryption algorithm to hash data but introduces a work function that determines how much work it will take to hash the data.

By setting a large value for the work function, you can make bcrypt take an arbitrarily large amount of CPU time (say 0.1 seconds on a modern system) to encrypt the password. This can obviously affect system performance (e.g., if you have 10 users logging in every second, all the CPU time would be consumed by bcrypt).

The advantage of bcrypt is that, as time goes on, you can increase the work function, defeating attackers in the future (assuming they didn't steal the password file 10 years ago). You might think a similar outcome could be achieved by using multiple rounds of MD5 or AES, but that would actually make the system easier to attack. For example, using 1,000 rounds of MD5 gives the attacker 1,000 possible values, which when MD5'ed 1,000 times will result in a password stored in their rainbow table. So, please don't do that.

A Note on Software

Of course, none of this matters if the software you are using doesn't work properly. A perfect example is the release of PHP version 5.3.7, which contained a critical security bug. A small change (to avoid a warning) was made to the `crypt()` function when using the MD5 algorithm (the default).

The result of this small change was that, instead of passing back the salt value and the password, `crypt()` concatenated the password to the salt (essentially making a large salt value with no password). Thus, if anyone else tries to login, only the salt values will be compared. The values will of course match, thus allowing the bad guy in. So, I guess the moral of the story is to run your unit tests when you make changes to cryptographic functions.

Conclusion

Brute-forcing older algorithms is definitely possible now (DES and 3DES already fell to brute-force attacks several years ago). The latest algorithms like AES and SHA are good, but, ironically, one of their biggest strengths, their speed, also works against them. So, choosing something slower like bcrypt might be a good idea.

Finally, it doesn't matter how you encrypt passwords if you allow users to pick weak passwords (especially words listed in dictionaries). You might want to download a few dictionaries and check against them when a user attempts to set or change a password. ■■■

INFO

[1] Accelerating SSL with GPUs: *http://www.ndsl.kaist.edu/papers/comm022t.pdf*

[2] OpenSSL-GPU: *http://labs.sasslantis.ee/openssl-gpu/*

[3] pgcrypto: *http://www.postgresql.org/docs/8.3/static/pgcrypto.html*

[4] Free Rainbow Tables: *http://www.freerainbowtables.com/*

[5] RainbowCrack Project: *http://project-rainbowcrack.com/*

[6] bcrypt: *http://bcrypt.sourceforge.net/*