Proper Password Hashing

Clearly A Paper We Must Write

@BallastSec http://ballastsec.blogspot.com/
Authors:
bwall (@bwallHatesTwits)
drone (@dronesec)
June 27th, 2012

1 Preface
    1.1 Current State
    1.2 Moral Obligation
2 Proper Hashing Methods
    2.1 Salt is good for you
    2.2 Key Derivation Functions
        2.2.1 PBKDF2
        2.2.2 ARC4PBKDF2
        2.2.3 bcrypt
3 Conclusion
    References and Further Reading

1 Preface

It is clear that we have to write this paper as almost countless services that we trust with our passwords are handling them irresponsibly. Hash crackers are extremely effective these days, to the point that most passwords are crackable within negligible time. Even long passwords aren't safe these days under a single hash. First we will look at the current state of things, then why your service shouldn't take this lightly, then the various ways you can improve your password storage.

1.1 Current State

As we have seen from various password database dumps recently, most websites do not properly store their users' passwords. Storing passwords in plaintext or with a single hash algorithm without a salt seems to be the commonplace. This is irresponsible. If you really want to save yourself milliseconds of computing time instead of properly protecting your users, then I will openly suggest against using your service personally.

1.2 Moral Obligation

If you in any way store anyone's passwords, you are liable for them. Every time someone hacks into someone's database and releases the plaintext passwords or weak hashes, all of those users not only need to change their passwords for every service they use that password for, but also need to seriously consider whether they want to continue using your service. If you don't secure these passwords properly, you will feel the consequences at some point. Take the time to properly generate and store the password hashes and the entropy.

2 Proper Hashing Methods

It is not enough to just hash the password. It must be hashed properly. Here are a few methods and guidelines.

2.1 Salt is good for you

Salt in the sense of hashes is metaphorically similar to salt in the sense of food. You add it to your hash for that extra bit of flavor. The only difference is, we want more than a bit of flavor with a salt used in a hash. Password salts are a type of entropy used to make hashes from the same algorithm with the same plaintext different. This is done by hashing the salt with the plaintext. A salt alone is not good enough to make a hash secure, but it is better than no salt at all.

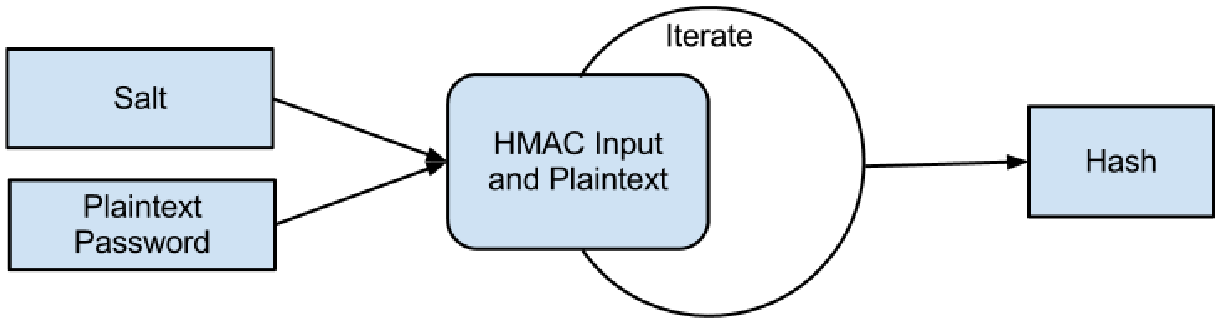
Salts are also used to increase the time complexity of dictionary attacks and rainbow tables. Now, not only does an attacker need to generate tables with potential passwords, they also need to generate tables with the various salts. This is why it is important that different salts (preferably random bytes) are used for every single hash; not only to ensure user passwords do not collide if they're equal, but to increase the complexity and size, and implicitly the difficulty, of a dictionary or brute-force attack. Do not repeat salts.

2.2 Key Derivation Functions

Interestingly enough, the best functions for hashing passwords were originally intended for generating keys for encryption from values not suitable to be keys. These functions process input in a cryptographic manner and output a set number of bytes that can be used as a key for encryption. They also happen to be a one way encryption method which take more processing than hashing algorithms, and can be easily configured to be stronger.

2.2.1 PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) is an RSA Laboratories key derivation function used to derive strong, hash-based keys. It works by applying a pseudo random hash function (such as SHA-256, SHA-512, etcetera) to a string, in our case a password, along with a salt and repeating the process many, many times. This process can be generalized with the following diagram:



The discussion of the salt has already been taken care of, though it is worth mentioning that the recommended salt length for a PBKDF is at least 128 bits. The PBKDF specification[1] states that SHA-1[2] is the approved PRF. However, SHA-1 as of 2005 is demonstrably weak[3]and should not be used as an HMAC function. It is in your best interest to choose an HMAC that is strong enough to withstand brute force, calculated attacks, and potential future design issues (the ‘breaking’ of the algorithm, or computational feasibility). It is important to note the difference between a hash and an HMAC -- a hash is not concerned with **message authentication**. It is only concerned with the integrity of the data. An HMAC is typically used to sign messages, hashing it with a salt that is unique to its owner. This can also be used to generate very strong hashes. The problem is that one HMAC does not take very much to process, and is still just as easy to brute force knowing the salt. That is where PBKDF2 comes in, implementing an iterative HMAC which continues to increase security and cracking time of each individual hash. Let's take a look at some pseudo code for this process:

```
bytes PBKDF2(HashAlgo, Plaintext, Salt, Iterations)
{
      bytes IterativeHash = hash_hmac(HashAlgo, Salt, Plaintext);
      bytes TempHash = IterativeHash;
      for(int iter = 0; iter < Iterations; iter++)
```

1 http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf

2 http://www.itl.nist.gov/fipspubs/fip180-1.htm

3 https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

```
	{
		TempHash = hash_hmac(HashAlgo, TempHash, Plaintext);
		IterativeHash ^= TempHash
	}
	return IterativeHash;
}
```

This is a simplified version of the full pseudo code. It is also modified properly to meet our needs optimally for using it as a hashing function, instead of being a key derivation function. Here is a PHP implementation of it:

```
function PBKDF2($plain, $salt, $iterations, $algo = 'sha256' )
{
	$derivedkey = $b = hash_hmac($algo, $salt . pack('N', $block),
$plain, true);
	for ( $i = 0; $i < $iterations; $i++ )
	{
		$derivedkey ^= ($b = hash_hmac($algo, $b, $plain, true));
	}
	return $derivedkey;
}
```

Though generalized, the above examples give you a basic look into how a PBKDF2 function can be implemented. It is as simple as hashing the salt and plaintext for DK, then iterating over the same function with the plaintext and previous hash (DK_{i-1}), and finally XORing the DK and previous DK. Doing this 1,000 or more times will generate a strong, slow to decrypt hash that you can use to securely store your passwords. Please refer to the referenced NIST-SP800-132 for more details.

2.2.2 ARC4PBKDF2

So I(bwall) have been toying around with the idea of dynamic entropy for a while now. PBKDF2 seemed like the good place to put it to work. The idea is to make entropy to an encryption algorithm which changes during the encryption process. This idea stems from the limitations of of some of the faster hash cracking systems who tend to highly optimize certain processes such as the application of entropy. So to combat this, we can add a generally fast process to create additional entropy that would require additional computation by the hash cracking process and possible removal of certain optimizations. In ARC4PBKDF2, an ARC4 encryption stream is initiated with a key, then that stream is used to encrypt the plaintext before it is used in the HMAC during every iteration, continuing the same stream, also encrypting the output from the HMAC. This same ARC4 stream must be maintained through the whole encryption process, which adds to the complexity of an already difficult hashing method to crack. Here is some sample code in C#.

```
public byte[] Hash(byte[] input)
{
  ARC4 rc4 = new ARC4(ARC4Key);
byte[] derived = new
HMACSHA256(Salt).ComputeHash(rc4.CryptBytes(input));
  byte[] temp = derived;
  for (int x = 0; x < Iterations; x++)
  {
temp = rc4.CryptBytes(new
HMACSHA256(temp).ComputeHash(rc4.CryptBytes(input)));
    for (int y = 0; y < derived.Length; y++)
    {
       derived[y] ^= temp[y];
    }
  }
  return derived;
}
```

It is important to note that in the ARC4 implementation, the CryptBytes method continues to use the same ARC4 stream, so each encryption is done at different parts of the stream. The dynamic entropy is a new experimental idea, which has yet to be contested by those who develop the optimization methods it's designed to defeat.

2.2.3 bcrypt

bcrypt is an adaptive hashing function introduced in 1999, and works a bit like PBKDF2, but in a more complicated manner. The introductory paper, published by Niels Provos and David Mazieres, can be found here and in great detail explains the subtle intricacies of the algorithm and its implementation. For our purposes, a brief overview followed by an example will suffice.

bcrypt is essentially the Blowfish[4] block cipher in ECB with a more complicated key scheduling algorithm, specifically with the S-boxes. This allows the algorithm to be future-proofed and substantially more adaptive. Underneath the covers, the implementation uses a 128 bit salt and the enhanced algorithm, known as *eksblowfish*, or expensive key schedule blowfish. The bcrypt function header looks like this:

```
bcrypt(cost, salt, pwd)
```

Where the cost is the key scheduling controller (i.e. how expensive is the key scheduling phase), the salt is a 128-bit value, and the password is the plaintext (up to 56 bytes) key, used for encryption in the Blowfish algorithm.

There are a couple of really wonderful, advantageous things to bcrypt that bear discussion. One is that the algorithm REQUIRES a salt; although salts alone will not save your stolen SQL dump, they will increase their complexity. The second is that the algorithm is, and I've used this term several times, adaptive. The key scheduling runtime can be as little as 1ms or as slow as you want. The wonderful thing about this is that a .3ms password check compared to a 3s password check is negligible for a server, but for an attacker this complexity increase is absolute chaos. This also directly combats with Moore's law, as over time you may gradually tweak the cost variable to increase the password complexity.

Bcrypt is one of the most popular key derivation functions (right next to PBKDF2) and nearly every language has a solid, foolproof implementation available. If you are interested in the

[4] https://www.schneier.com/paper-blowfish-fse.html

advanced key scheduling algorithm, I recommend the official paper above. It is out of the scope of this paper, and for our purposes a small function example implementing basic features will suffice:

```
public string bcrypt(int cost, byte[] salt, byte[] password)
 {
    byte[] state = EksBlowFishSetup(cost, salt, password);
    string ciphertext = "OrpheanBeholderScryDoubt";
    for (int i = 0; i < 64; ++i)
    {
        ciphertext = EncryptECB(state, ciphertext);
    }
   return cost.ToString() + BitConverter.ToString(salt) + ciphertext;
 }
```

The above code is fairly straightforward. The initial state is set up with the EksBlowFish algorithm; this stage is the most time-consuming of the stages. Following this is the encryption of a 192-bit magic string, almost always `OrpheanBeholderScryDoubt` in ECB with its previous state and the resulting EksBlowFish result. The final output is a static 192-bit concatenation of the cost, salt, and resulting ciphertext.

3 Conclusion

These methods significantly improve the time required to crack hashes. There is no excuse not to use methods mentioned in this paper to store passwords. We understand that handling hashes might be difficult, especially if you don't really understand them. We at Ballast Security are working on a solution for this, so even those who don't have a clue of how to secure passwords will be able to do so with minimal effort.

References and Further Reading

[0] Bruce Schneier's description of the BlowFish cipher
https://www.schneier.com/paper-blowfish-fse.html
[1] Niels Provos' and David Mazieres original paper on Bcrypt
http://static.usenix.org/events/usenix99/provos.html
[2] NIST SP800-132: Recommendation for Password-Based Key Derivation
http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf
[3] FIPS 180-1 on Secure Hashing
http://www.itl.nist.gov/fipspubs/fip180-1.htm