PolyPasswordHasher

Improving Password Storage Security

SANTIAGO TORRES AND JUSTIN CAPPOS

Justin Cappos is an assistant professor in the Computer Science and Engineering Department at New York University. Justin's research philosophy focuses on improving real world systems, often by addressing issues that arise in practical deployments. jcappos@nyu.edu

Santiago Torres is a graduate student in the Computer Science Department at New York University working under Justin Cappos's mentorship. He is currently a contributor to open source projects such as "TUF," a secure update framework, and "PolyPasswordHasher," a password storage mechanism resistant to cracking. santiago@nyu.edu

We most often hear about password database thefts and the subsequent cracking of these databases' hashed passwords. Since systems have become faster, and attackers have gained access to clusters or specialized hardware used for cracking, the techniques that have made cracking difficult need to be updated. We have created a system, PolyPasswordHasher, that uses shared keys to add an additional encryption step; it requires an attacker to simultaneously crack several keys at once. We project that PolyPasswordHasher changes the time needed to crack even short passwords to longer than current estimates of the age of the universe.

The Current Standard in Password Protection

Initially, passwords were stored in plaintext on servers. However, once a password database was stolen by an attacker, all passwords on the system could be read. To combat this, password storage systems started to store a cryptographic (one-way) hash of a password. In this scheme, after acquiring a password database, the attacker had to guess at passwords and check their values against the stored hashes in order to recover the actual passwords.

Cracking cryptographic hashes is not as complicated as it sounds, because an attacker can simply pre-compute a database of common passwords and look up a password when given its hash. To address this flaw, "salting" was devised; salt is a random value that is used in the cryptographic hash of the password to make it effectively unique, per database. Current best practice is to create a unique salt for every password (stored alongside the cryptographic hash in the database).

How Do Hackers Steal and Crack Passwords?

To log in, a user provides his or her login name and password to the server. If the user is remote (not physically at the server), this is done over an encrypted channel so that a man-in-the-middle cannot see the user's password. The server receives the user's password, performs a secure salted hash, and checks it against the value stored in the database. If these match, the user is allowed to log in.

When an attacker wants to steal the password for a certain account, there are three options: obtain the password before it gets hashed, act as a man-in-the-middle, or acquire the hash and crack the database. Getting a password before it gets hashed requires the ability to read arbitrary memory (root access) on a running server. Attacks of this nature, in which the server has been completely compromised, account for less than 5% of total compromises, according to Mirante's analysis of recent password hacks [3].

Attacks that try to acquire the password while in transit (as a man-in-the-middle) are even less common. The attacker must both intercept the client's traffic and fool the user into thinking the attacker's site is in fact the actual site they are attempting to log in to. While not perfect, technologies such as SSL and HSTS make thefts that use this technique uncommon.

The most popular method is for the attacker to obtain a copy of the hashed password database. This commonly occurs when a copy of the hashed password database (e.g., a backup disk) is lost. Attackers also can trigger a hashed password database disclosure, with SQL injections accounting for the majority of known password database breaches.

A hacker who gains access to a hashed password database will usually try to crack passwords on a remote system (offline) by guessing and computing passwords' stored hashes, looking for a match. Cracking programs such as oclHashCat [4] or John the Ripper [2] can automate this process. To give this some perspective, a dump of passwords for 60% of the 6.5 million stolen LinkedIn accounts was found one week after the breach on a hacker forum. This is perhaps not surprising since a security researcher was able to crack 63% of a ~40,000 entry salted SHA1-encoded database in 40 minutes. Given this state-of-affairs, salted password hashes are not a sufficient protection strategy.

A New Defense Scheme: PolyPasswordHasher

To meet the need for enhanced password security, we have created PolyPasswordHasher, a password storage scheme that makes stored password hash data (called polyhashes) interdependent and thus impossible to crack individually. An attacker that obtains a password database stored using PolyPasswordHasher must crack groups of passwords *simultaneously*. The principle that makes this work is the concept of *cryptographic shares*, such as in a Shamir Secret Store [1, 5].

Imagine these cryptographic shares functioning something like a "two-man rule," such as when a bank check requires multiple signatures or two physical keys must be turned at the same time to open a safety deposit box. A secret key is divided into multiple pieces of information, called shares, with each piece distributed across at least two keyholders. This share strategy aids in the process of recombination. When a certain number of these pieces of information are acquired, an agent is able to recover the original secret key. One important characteristic is that if an agent has only some of the pieces of information needed, they recover no information about the original secret key.

The principal characteristic of this sharing scheme is a configurable *threshold value*, usually set to a value such as 3 or 5, which determines how many shares are needed in order to recover the secret key. The secret key is never stored on disk by PolyPasswordHasher to secure it from attacks such as SQL injection. Instead of storing a secure salted hash, PolyPasswordHasher stores a different value, called a *polyhash*. A polyhash consists of the secure salted hash for the password, XORed with a cryptographic share. This protects a password's secure salted hash with the cryptographic share. That is, before individual passwords can be cracked, an attacker must be able to recover the secret key (recoverable via a threshold of passwords).

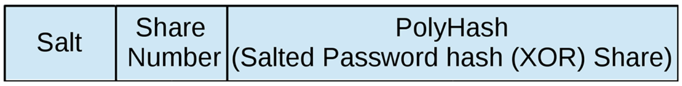

Figure 1: How a polyhash is stored for a threshold account

In the following sections, we first describe normal operation of a PolyPasswordHasher server (by assuming that a server has a threshold of passwords, and thus the secret key). We then discuss how a system using PolyPasswordHasher bootstraps after a reboot.

How PolyPasswordHasher Works When a Threshold of Passwords Is Known

PolyPasswordHasher supports two types of user accounts: those that protect a cryptographic share (threshold accounts) and those that do not (thresholdless). Types of accounts that would not protect a share are those in which users are allowed to register any number of accounts, as is the case with Gmail or Facebook. Whether accounts are threshold or thresholdless is invisible to the user, with different procedures taking place in the background.

When a threshold account is created, the system produces a random salt, calculates a salted-hash and issues a new share. The system produces a polyhash by XORing the salted hash and the share, which is then stored, along with the salt and some helper information, as illustrated in Figure 1. The share itself and the salted password hash are never stored on disk.

To log in, a user gives his or her username and password to the server. PolyPasswordHasher checks these to identify which share was assigned to the user's polyhash and then recomputes that share. Next, a salted-hash will be calculated from the input password and its stored salt. Finally, the newly created salted-hash will be XORed with the share to construct a polyhash. Assessing whether the user provided the correct password is a matter of checking the constructed polyhash against the stored polyhash.

If in addition to threshold accounts the system allows other users to freely create accounts (e.g., Gmail), a *thresholdless* entry will be issued for those users. Instead of assigning a share, the secure salted-hash for a thresholdless entry is encrypted with the secret key. Verifying an account for this new user entails decrypting the stored encrypted hash and comparing it in the same fashion as are regular salted-hashes; thresholdless entries are illustrated in Figure 2.

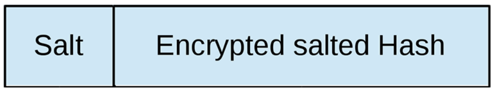

Figure 2: Stored data for thresholdless accounts

Bootstrapping a Server after Reboot

A PolyPasswordHasher server stores its secret key in memory, not on a disk, and the key is thus lost upon reboot. When the server reboots, this secret key is not available, and thus the server cannot compute shares. Therefore, PolyPasswordHasher cannot verify or create accounts as it normally does. PolyPasswordHasher must *bootstrap*.

During this phase, PolyPasswordHasher will collect shares from threshold logins in order to recover the secret. The number of threshold logins required to recover the secret is configured by the system administrator, and it is usually set to a low value (e.g., three or five). For example, if the threshold is three, PolyPasswordHasher will finish bootstrapping after the third threshold account has provided a correct password. While PolyPasswordHasher waits for threshold accounts to log in, it authenticates user passwords using a field called *partial-bytes*.

The partial-bytes field contains only a portion of a regular salted-hash, such as the last four bytes. When a user attempts to log in during the bootstrap phase, PolyPasswordHasher will verify that the partial-bytes field matches the corresponding portion of the password's secure salted hash. For example, if the last four bytes of the salted hash are "A04F," then this will be verified upon login. Although these partial-bytes could *hint* to the attacker what the user's password is, the attacker would not be certain of the password since the complete salted hash is not stored. If the attacker chooses a password that matches the partial-bytes but nonetheless is incorrect, this will be detected after bootstrapping is finished, and the system administrator notified of the likely password hash database theft.

Account creation is also available during the bootstrap phase. To enable this, the new account is added to the database with a regular salted-hash. These accounts can be used normally while the system is bootstrapping. When the system is provided shares from enough threshold accounts, it can finish bootstrapping. To do this, the server re-validates all prior logins with the full polyhash or encrypted salted-hash. Also, any accounts that were created during bootstrap will have their password hash transitioned to protected shares (if threshold) or encrypted shares (if thresholdless).

Evaluation—How We Know It Works

Three elements contribute to the effectiveness of a new password storage method: overhead (e.g., storage and memory costs), efficiency, and time to crack passwords. We assessed storage costs by analyzing the amount of extra information that is required by PolyPasswordHasher and compared that with a standard user database. The only additional information required is the *share number* field and the *partial-bytes* field. The share number requires one extra byte per entry, and the partial-bytes requires four bytes, although this last value is configurable. The total extra information required is, then, five bytes per entry. Considering that the salt, username, and salted-hash fields account for more than a hundred bytes per entry, we expect the overhead to be less than 5% of the password database storage space cost. Furthermore, the size of a hashed password database is minimal compared to user data (photos, content, etc.) on most systems.

The memory cost of an implementation consists only of a buffer to hold the secret. The size of the buffer for the secret key ranges from 16 bytes to 64 bytes, depending on the implementation.

To understand the instruction efficiency (performance) of PolyPasswordHasher, we performed a series of microbenchmarks on an early 2011 MacBook Pro with 4 GB of RAM and a 2.3 GHz Intel Core i5 processor using a Python reference implementation. We measured instruction efficiency by looking at the time it took for different operations of the PolyPasswordHasher algorithm to complete. We found that the algorithm takes about 150 microseconds to authenticate a user. To transition from the bootstrap phase to normal operation, which is only done once upon restart, takes between hundreds of microseconds to tens of milliseconds after the last threshold account has provided a correct password, depending on the threshold value.

Suppose that users choose passwords from one of the 95 easily typeable characters. If users choose six-character, random passwords, there are only $7.35*10^{11}$ possible values. When stored with PolyPasswordHasher and a threshold of three, an attacker would need to search $3.97*10^{35}$ different combinations—more than 23 orders of magnitude more operations.

To put these numbers into perspective, using the best known GPU-cracking techniques, a computer can compute about one billion hashes per second [6]. If three passwords were stored with salted hashes (not PolyPasswordHasher), there are $3*7.35*10^{11}$ combinations possible. It would take an attacker less than an hour to try these combinations on a single computer. With PolyPasswordHasher, to search the keyspace of $3.97*10^{35}$ combinations would take all 900 million computers on the planet $1.39*10^{10}$ years. That is longer than the estimated age of the universe.

Summary / What's to Come

There are multiple, open source implementations of PolyPasswordHasher available. Our Django implementation for PolyPasswordHasher is currently being integrated into a variety of servers at New York University. We will use data from these servers to help us understand whether there are any unforeseen complications with production use.

We invite interested parties to find out more information and try out PolyPasswordHasher at: http://polypasswordhasher.poly.edu.

References

[1] K. Hirokuni, "Divide and Manage Secret Data Securely with Shamir's Secret Sharing—Kim's Tech Blog": http://kimh.github.io/blog/en/security/protect-your-secret-key-with-shamirs-secret-sharing/.

[2] John the Ripper official Web site: http://www.openwall.com/john/.

[3] D. Mirante, J. Cappos, "Understanding Password Database Compromises," Polytechnic Institute of NYU, Department of Computer Science, Technical Report TR-CSE-2013-02 9/13/2013.

[4] oclHashcat official Web site: http://hashcat.net/oclhashcat/.

[5] "Shamir's Secret Sharing," Wikipedia: https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing.

[6] A. Zonenberg, "Distributed Hash Cracker: A Cross-Platform GPU-Accelerated Password Recovery System," Rensselaer Polytechnic Institute (2009).