FAQ on Honeywords

Ari Juels and Ronald L. Rivest
May 2013

Question 1: What are honeywords?

Answer: Honeywords are a defense against stolen password files. Specifically, they are bogus passwords placed in the password file of an authentication server to deceive attackers. Honeywords resemble ordinary, user-selected passwords. It's hard therefore for an attacker that steals a honeyword-laced password file to distinguish between honeywords and true user passwords. ("Honey" is an old term for decoy resources in computing environments. To the best of our knowledge, the term "honeywords" was coined in this paper.)

For example, one of the following passwords is a true user password. The rest are honeywords (generated using the simple "chaffing-with-a-password" algorithm in our paper). Which is the real password?

```
kebrton1
02123dia
a71ger
forlinux
1erapc
avanture32
sbgo864959
aiwkme523
```

An attacker that has stolen a password file may crack its hashed passwords (see Questions 4 and 5 below) and attempt to impersonate users. Given the presence of honeywords, though, such an attacker is unlikely to guess a user's true password and likely instead to submit a honeyword. If a honeyword-enabled system detects an attempt to login using a honeyword, it raises an alarm indicating that the password file has been compromised.

Honeywords aren't visible to users and don't in any way change their experience when they log in using passwords.

Question 2: Are stolen password files a big problem?

Answer: In April 2013, LivingSocial had about 50 million user passwords stolen and published in hacker forums. Evernote lost a similar number in the previous month. Other prominent password breaches have taken place at Zappos (24+ million passwords), LinkedIn (6+ million passwords), and RockYou (32+ million passwords). And these are just the publicly reported incidents!

Password compromise also seems commonly to play a role in silent, targeted attacks. For example, attackers infiltrating the *New York Times* used password cracking to compromise accounts.

Question 3: How is the use of honeywords by an attacker (and thus stolen passwords) detected?

Answer: An auxiliary service called a *honeychecker* checks whether a password submitted by a user on login is her true password or a honeyword.

The password system itself stores a given user's password randomly along with honeywords. Together, the password and honeywords are called *sweetwords*. The user's password is placed within a sweetword list at a random position c, which is stored by the honeychecker. The password system itself doesn't know which sweetword in the list is actually the password.

When there's an attempt to log into the user's account with a sweetword, the password system sends the position j of this sweetword in the stored list to the honeychecker. The honeychecker verifies that j = c, i.e., the true password has been submitted; otherwise it raises an alarm, as a honeyword has been submitted.

If an incorrect password is submitted that is also not a sweetword, the system uses its ordinary policy for such cases.

Question 4: What is password hashing? Isn't it already a common way to protect passwords?

Answer: Password hashing is the application of a cryptographic one-way ("hash") function H to a password P. The resulting value H(P) is stored in a system's password file in lieu of the raw password. To check the correctness of a submitted password P', the system computes H(P') and verifies that H(P') = H(P).

The benefit of hashing is that it helps conceal the password P should the password file be stolen. If the password P is well chosen (and H a good hash function), it is not straightforward to extract P from H(P). The only way to determine P is by brute force, meaning hashing password guesses P' until one is found such that H(P') = H(P).

To prevent compilation of dictionaries of pre-hashed passwords under H and reduction of brute-force guessing to a dictionary lookup, an enhancement called "salt" is common. What is actually stored for a user account is the pair (H(P,S),S), where S, the salt, is a random, per-user value (typically 128 bits or so), and H(P,S) is a hash of both the password and the salt.

It is common for computer systems to store hashed passwords, rather than raw ones. Honeywords may be used with any form of password hashing: Honeywords may be hashed along with true passwords. (We strongly recommend the use of hashing and salt for passwords in any system.)

Question 5: Why isn't hashing good enough to protect passwords?

Answer: Users tend to pick passwords that are easy for an attacker to guess, in the sense that they're common choices or simple variants on common choices. One popularly selected user password, for instance, is 123456 (chosen by almost 1% of users in one popular service!).

The process of extracting a password P from H(P) is called "password cracking." Given today's generation of hash functions, cracking passwords is relatively easy. It requires little expertise: There are common, user-friendly tools (John the Ripper, Hashcat) for cracking hashed passwords. And because users tend to choose poor passwords, cracking them often doesn't require a lot of computation.

It is possible to design a hash function H that requires more computation to compute than a standard one and thus more computation to crack. (Bcrypt is an example.) But because a password system needs to apply a hash function to verify a submitted password, H can't be so slow that it creates a delay noticeable to users. And as long as users choose poor passwords, the number of guesses required by an attacker to crack many of them will remain relatively small. If 1% or so of users select 123456 as their password, then an attacker can crack at least one user's password with only about 50 tries on average!

Additionally, given hashing alone, there is no way, as with honeywords, to *detect when a password file has been stolen*.

Question 6: What happens if an attacker compromises the password system AND the honeychecker?

Answer: An attacker that compromises both elements of the system (and cracks hashes) can learn users' passwords. The honeywords approach is designed so that the defender operating the password system is *no worse off in this case than without honeywords*.

Question 7: Why is it any harder for an attacker to compromise the honeychecker than the password system?

Answer: The password system and honeychecker are independent systems. They can be placed in separate administrative domains, run different operating systems, and so forth. This separation of duties helps harden the system as a whole, as it

means that an attacker must separately compromise both elements to obtain users' passwords. (In general, Honeywords inherit the security benefits of "distributed cryptography.")

Additionally, the honeychecker can be an especially simple system operating downstream of the password system in a highly protected environment, such as a security operations center. In can even be deployed such that it does not produce outputs other than alarms.

Question 8: How are honeywords generated?

Answer: Our paper discusses a number of approaches. Some involve "tweaking" a real password to generate similar-looking alternatives. Others involve synthetic generation based on published lists of users' passwords, and thus, indirectly, on the password-composition habits of ordinary users.

There are countless alternatives. Improving honeyword generation and storage is a very interesting research problem, we believe.

Question 9: How many honeywords per user should be stored? Don't honeywords create storage overhead?

Answer: The number of required honeywords depends on the quality of the honeywords and the desired strength of the system against an attacker. Even a small number of well-selected honeywords (say, 10 per user) is sufficient to detect abuse of a stolen password file with very high probability after just a small number of adversarial login attempts—or to deter an attacker from trying to impersonate users.

As for storage overhead, many password systems already store ten or more old passwords to ensure against users repeating old passwords too frequently. Honeywords merely use a comparable amount of storage. Our paper also discusses some tricks for reducing storage overhead.

Question 10: What happens if a user accidentally types a honeyword?
Answer: Honeywords may be crafted to minimize the likelihood of this problem. For example, for the sweetword list in Question 1, such accidents are improbable.

Question 11: Are there stronger approaches than honeywords to protect passwords?

Answer: Yes. For example, RSA, The Security Division of EMC, sells a product called Distributed Credential Protection (DCP). (One of us helped develop it.) DCP splits

passwords across two servers, such that if either server is compromised—or both at distinctly different times—*no* information about passwords is revealed. DCP implements what is called "distributed cryptography." Honeywords may be viewed as a kind of lightweight distributed cryptography, simpler to deploy than DCP, but not as strong.

Question 12: Aren't there stronger forms of authentication than passwords? Ideally, shouldn't we be moving to such alternatives rather than using passwords in any form—even with honeywords?

Answer: Yes.

Question 13: Is there code available for a working honeyword system?

Answer: At this point, the scheme exists only on paper. We would warmly welcome efforts by the community to standardize honeywords, create references architectures, and/or make implementations of honeywords freely available.

Question 14: Is there a licensing fee for honeywords?

Answer: There's no licensing fee, and we're unaware, at least, of any patent encumbrance on honeywords as proposed in our paper. Our hope is to see the technique used as widely as possible against a serious and mounting threat. (See the IP statement in the paper for more details.) You do not need to contact us to make use of the techniques in our paper.