Honeywords: Making Password-Cracking Detectable

Ari Juels
RSA Laboratories
Cambridge, MA, USA
ari.juels@rsa.com

Ronald L. Rivest
MIT CSAIL
Cambridge, MA, USA
rivest@mit.edu
ABSTRACT
Honeywords

- Simple method
- Improving security of hashed passwords

Password or Honeyword?

Use of honeyword
INTRODUCTION
Passwords are weak

- Users frequently choose poor passwords
- Adversary applies brute-force attack

Real passwords are often weak and easily guessed.

Top 10 passwords

- 123456 = 1666 (0.38%)
- password = 780 (0.18%)
- welcome = 436 (0.18%)
- ninja = 333 (0.08%)
- abc123 = 250 (0.06%)
- 123456789 = 222 (0.05%)
- 12345678 = 208 (0.05%)
- sunshine = 205 (0.05%)
- princess = 202 (0.05%)
- qwerty = 172 (0.04%)
How about an example?

- October 2013
  - Adobe lost 130 million passwords

- March 2013
  - Evernote lost 50 million passwords

- July 2012
  - Yahoo lost 130 million passwords

- June 2012
  - Linkedin lost 130 million passwords
Can we tighten security?

- Make password hashing more complex and time-consuming
  - Improve password security
  - Slow down legitimate user’s authentication
  - Doesn’t make successful password cracking easier to detect
Fake user accounts

- **Honeypots**
  - Help to password cracking detection
  - Adversary can distinguish fake accounts
    - Usernames
    - Account’s activity

[Diagram showing login, raise alarm, and honeypot used]
Paper’s approach

- Extending previous idea for all users
  - Multiple possible passwords per user
  - Set off an alarm if a honeyword is triggered

- Makes password cracking detection easier
- Effective and easy to implement
- Useful layer of defense

Alice

Honeyword1
Honeyword2
Honeyword3
Real_password
Honeyword4
Terminology
Terminology

Honeywords

Hashed Passwords' File

Sweetwords

Alice:

\[ P_1 \]

\[ P_2 \]

\[ ... \]

\[ P_i = P \]

\[ ... \]

\[ P_n \]
Attack scenarios

- Stolen files of passwords hashes
  - offline brute-force computation
- Easily guessable passwords
  - poorly or common passwords
- Visible passwords
- Same password for many systems
- Passwords stolen from users
  - phishing
- Password change compromised

➔ We focus on the first attack scenario
  - Adversary has file of usernames and associated hashed passwords
Honeychecker
Honeychecker

What is it?

- An auxiliary secure server
- Communication is over dedicated lines and/or encrypted and authenticated
- Capable of taking an action
Honeychecker

- Maintain a single database value for each user

<table>
<thead>
<tr>
<th>Users-Real password pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Table C</strong></td>
</tr>
<tr>
<td>Alice</td>
</tr>
<tr>
<td>Bob</td>
</tr>
<tr>
<td>Jax</td>
</tr>
<tr>
<td>Tommy</td>
</tr>
</tbody>
</table>
Honeychecker API

<table>
<thead>
<tr>
<th>Alice</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>3</td>
</tr>
<tr>
<td>Jax</td>
<td>1</td>
</tr>
<tr>
<td>Tommy</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table c**

Secure Channel

Set(i,j): Sets c(i) to have value j

Set(2,4)  

Check(i,j): Checks that c(i) = j.

Check(2,4)  

Alice 4
Bob 4
Jax 1
Tommy 1
Honeychecker

Design principles

- Extremely simple
- Minimal amount of secret state
- Little overhead in computation and communication
- The compromisation of the honeychecker at worst only reduces security to the level it was before honeywords and honeychecker was introduced, since it only stores random small integers.
Login
Every time someone tries to login:

**System's Database**

- **Alice**
- **Bob**
- **Jax**
- **Tommy**

- **Honeyword1**
- **Honeyword2**
- **Honeyword3**
- **Real_password**
- **Honeyword4**

**User-Password**

<table>
<thead>
<tr>
<th>User</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
</tr>
<tr>
<td>Jax</td>
<td>1</td>
</tr>
<tr>
<td>Tommy</td>
<td>1</td>
</tr>
</tbody>
</table>

**Check(1,4)**
Every time someone tries to login:

- Honeyword1
- Honeyword2
- Honeyword3
- Real_password
- Honeyword4

System’s Database:

- Alice
- Bob
- Jax
- Tommy

User-Password:

<table>
<thead>
<tr>
<th>User</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
<tr>
<td>Bob</td>
<td>3</td>
</tr>
<tr>
<td>Jax</td>
<td>1</td>
</tr>
<tr>
<td>Tommy</td>
<td>1</td>
</tr>
</tbody>
</table>

Take an action (determined by policy)
If password is neither the real one nor one of the user’s honeywords, login is denied!

**Actions**

- Notify administrator
- Let login proceed as usual
- Let login proceed on a honeypot system
- Trace the source of the login
- Turn on additional logging of the user’s activities
- Shut down user’s account
- Shut down the whole system
Change Password
Change password

- Create a new list of sweetwords (honeywords + real password)
  \{'New\_honeyword1, New\_honeyword2, New\_honeyword3, New\_password, New\_honeyword4\}
- Securely notify the honeychecker of the new real password’s index in sweetwords
- Update the user’s entry in system’s file

Current password

Don’t know your password?

New password

Confirm new password

Change Password Cancel

<table>
<thead>
<tr>
<th>Alice</th>
<th>New_Honeyword1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>New_Honeyword2</td>
</tr>
<tr>
<td>Jax</td>
<td>New_Honeyword3</td>
</tr>
<tr>
<td>Tommy</td>
<td>New_password</td>
</tr>
<tr>
<td></td>
<td>New_Honeyword4</td>
</tr>
</tbody>
</table>
Honeyword Generation
Honeyword Generation

- User’s password must be indistinguishable from honeywords

Which is Alice’s real password?

Alice:

- QrMdmkQt
- AP9LXEeA
- m7xnQVV4
- kingeloi
- y5BJKWhA

- How can we ensure that an adversary will not find the real password?
Approaches

Is there an impact on the user interface (UI)?

- **Legacy-UI**
  - Password-change UI is unchanged
  - User chooses his password

- **Modified-UI**
  - Password-change UI is changed for a better honeyword generation
  - User’s new password is modified
Chaffing by tweaking

- Chaffing-by-tail-tweaking
  - “Tweak” last t character positions

- Chaffing-by-tweaking-digits
  - “Tweak” last t positions including integers

Chaffing with a password model

- Honeywords could be real passwords
  - Take from published list

- Honeywords use password’s syntax
Chaffing-by-tail-tweaking

“Tweak” last \( t \) character positions

Let \( t = 3 \):

User-supplied password: \( \text{BG+7y45} \)
Chaffing-by-tweaking-digits

“Tweak” last t positions including integers

Let t = 2:

User-supplied password:

42*flavors

- 57*flavors
- 18*flavors
- 21*flavors
- 42*flavors
Chaffing-by-tail-tweaking

“Tweak” last $t$ character positions

Let $t = 3$:

User-supplied password: 42*flavors

- 42*flavrbn
- 42*flavctz
- 42*flavrew
**Tough Nuts**

- **What is it?**
  - Very hard password that the adversary will not be able to crack

  9,50PEe]KV.0?RIOt&L:-!J"b+Woi<*[!NWT/pb

  ➔ Give additional reason to:
  - Pause before diving in
  - Trying to log in with one of the cracked ones
Honeywords could be real passwords

| kebrton1   | 02123dia  |
| a71ger    | forlinux  |
| 1erapc    | sbgo864959|
| aiwkme523 | aj1aob12  |
| 9,50PEe]KV.0?RIOtceL-l:|J"b+WoI<*[!NWT/pb |
| xyqi3tbato| a3915     |
| #NDYRODD_!!| venlorhan |
| pizzhemix01| dfdhusZ2   |
| sveniresly | 'Sb123    |
| mobopy    | WORFmghtness|

Use a Published List

➢ List may also be available to the adversary

“Tough Nut”
Honeywords use password’s syntax

User-supplied password → Mice3blind → W4 | D1 | W5 →

- Bold3wings
- Gold5rings
- Hall2trick
- Goal0leaks
- **Take-a-tail**
  - Randomly chosen from the system
  - Required in the user-entered new password

- **Passwords randomly chosen by the system**
Take-a-tail

Sign Up
Already a member? Log In

Alice
myPassword

Propose a password: myPassword
Append “413” to password.
Enter new password: myPassword413

Generated honeywords:
myPassword798
myPassword982
myPassword113
myPassword056
myPassword935
myPassword664
VARIATIONS AND EXTENSIONS
‘Random pick’ honeyword generation

Generate a list of $k$ distinct random sweetwords

Example $k = 6$:

4Tniners, all41&14all, i8apickle, sin(pi/2), \{1,2,3\}, AB12:YZ90

Pick one element at random to be the new password (e.g. ‘AB12:YZ90’);

The other are the honeywords

Sweetwords can be generated by:

- The user
- An algorithmic password generator

This method is completely **flat, no matter** how we generate the passwords

**Which do you think is a better way of generating the sweetwords?**

**Why?**
Typo-safety

Rare for the user to set of an alarm by accident password == 'gt79' and honeywords == ['gt76', "gt77", "gt78", ...]
tail-tweaking requires the password tail to be quite different from the honeywords’ tails!

Honeywords’ tails should be quite different from each other as well.
Typo-safety
(example)

Example of using an error-detection code to detect typos

Use an error-detection code to detect typos! How? (example $t=3$)

Pick a small prime greater than 10: $q = 13$

tail$_2 = 913$  
tail$_1 = 413$

$3 \cdot (9) + 2 \cdot (1) + 1 \cdot (3) =$  
$3 \cdot (4) + 2 \cdot (1) + 1 \cdot (3) =$

$= 27 + 2 + 4 = 33$  
$= 12 + 2 + 3 = 17$

$|17 - 33| = 16$ The difference between these 2 should be a multiple of $q$. Here it is not, so... (#sorrynotsorry)

This property:
- is easy to arrange between sweetwords
- allows detection of any **single digit substitution** (e.g. 413 and 913)
- allows detection of **transposition of two adjacent digits** (e.g. 413 and 431)

Proof:
$$\text{err(tail}_1) - \text{err(tail}_2) = 3x + 2y + 1z - 3k - 2y - 1z = 3x - 3k = 3(x-k)$$

which will never be a prime, no matter the index

$$\text{err(tail}_1) - \text{err(tail}_2) = 3x + 2y + 1z - 3y - 2x - 1z = 3x - 2x + 2y - 3y = x - y$$

which will always be $< 10$, where $x, y$ are single digits
Managing old passwords

Many systems keep old passwords of users stored (usually the last 10).

**Prohibiting** a user from **reusing** her old passwords

Why do the authors **disagree** with this method?

- Hashes of old passwords should not be stored because **hashes can be inverted** on weak passwords
- A user has probably changed her passwords just because it was weak, but she may be **using on other systems**

*HER ACCOUNT ON OTHER SYSTEMS IS AT RISK*
Managing old passwords: authors’ suggestions

Record previously used password across the full user population

- A newly created password should not conflict with any of the passwords in the list (of previously used passwords)
- This list could be stored as a Bloom filter (not the hashed passwords themselves) for more efficiency

However..., if it required to store the old passwords

- In a protected module separated from the main system (distributed security), or ...
- Store them in the main system for legacy compatibility but,
  - encrypted
  - keys for encryption/decryption stored in the honeychecker
Storage optimization

Reduce storage of honeyword generation methods

Password = ‘32flavors’ then $T(password) =$

- Save a **random** on the computer system (e.g. **32flavors**)
- Save the index of the real password to the honeychecker (e.g. $C(i) = 33$, index of ‘32flavors’)

Example: Adversary or user submits a guess ‘g’ to the system for logging in (e.g. 67flavors)

- **Produce** $T(g)$ (e.g. $T(g)$ will be equal to $T(password)$)
- if $H(45flavors)$ in $T(g)$ then find the **index** of $g$ in $T(g)$
- if index == 45 ‘**ALARM**’
  - else if index == 33 ‘**allow login**’
  - else ‘**deny login**’
**Hybrid generation methods**

Combine the benefits of different honeyword generation methods

**chaffing-by-tweaking-digits** with **chaffing-with-a-password-model**

Password provide by user ‘abacad513’

**chaffing-with-a-password-model**

\[
\text{abacad513} \Rightarrow W_5 | D_{9,D}
\]

produce

<table>
<thead>
<tr>
<th>abacad513</th>
<th>snurfle672</th>
<th>zinja750</th>
</tr>
</thead>
</table>

**chaffing-by-tweaking-digits**

<table>
<thead>
<tr>
<th>abacad513</th>
<th>snurfle672</th>
<th>zinja750</th>
</tr>
</thead>
<tbody>
<tr>
<td>abacad941</td>
<td>snurfle806</td>
<td>zinja802</td>
</tr>
<tr>
<td>abacad004</td>
<td>snurfle772</td>
<td>zinja116</td>
</tr>
<tr>
<td>abacad752</td>
<td>snurfle091</td>
<td>zinja649</td>
</tr>
</tbody>
</table>
POLICY CHOICES
Password Eligibility

Some words may be ineligible as passwords.

Which passwords should not be used!

1. Password syntax
   a. **minimum length** (‘Hi’ can’t be a password)
   b. **minimum number of digits** (e.g. ‘myname41’ - for honeywords to be produced ‘myname42’, ...)
   c. **minimum number of special characters**

2. Dictionary words (‘giraffe’, ‘floWer’, etc.)

3. Most common passwords

#funfacts
The 20 most common passwords made up more than 10% of the surveyed passwords

The most common password "123456", makes up 4%
Failover

Logins should **proceed** even if the honeychecker has failed

Buffer messages on the computer system for later delivery to the honeychecker
Per-user and Per-sweetword Policies

Policies that vary per-user

Per-user policies

- **Honeypot accounts**: known only to the honeychecker
- **Selective alarms**: raise an alarm for sensitive accounts (administrator accounts)

Per-sweetword policies

- Hits on honeywords with *small edit distance* to the password should invoke a *less severe* reaction
  - To prevent user-typos
- Examples of such actions:
  - “Raise silent alarm,”
  - “Allow login,”
  - “Allow for single login only,” etc...
ATTACKS
General password guessing

Do not use common passwords

Take-a-tail method can reduce the probability of guessing the password by a factor of 1000

Example:
password = applethief

take-a-tail with t = 3
password = applethief355

- hard to remember
- can be brute forced in ms if you find ‘applethief’ in a dictionary
Targeted password guessing

Personal information **help** an adversary distinguish the password from the honeywords

Guess above user’s password from the list

<table>
<thead>
<tr>
<th>Honeyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>lovemycat45</td>
</tr>
<tr>
<td>lovemydog24</td>
</tr>
<tr>
<td>hatemyhamster87</td>
</tr>
<tr>
<td>ilikebeers64</td>
</tr>
<tr>
<td>stanley49</td>
</tr>
</tbody>
</table>

How can an adversary find personal information about the user?
Attacking the honeychecker

An adversary may decide to attack the honeychecker or its communications.

Requests to the honeychecker and replies from the honeychecker should always be authenticated!
Having stolen file F, calculate the probability of each sweetword being a honeyword or a password:

The probability that sweetword x is a password:

$$R(x) = \frac{U(x)}{G(x)}$$

U(x) user picked
G(x) algorithm generated

Example: ‘NewtonSaid:F=ma’

obvious structure to a human

not very obvious to an automatic generator
Denial-of-service

Methods such as chaffing-by-tweaking e.g. 45flavors, 46flavors, 47flavors, etc.

Give the opportunity to an adversary that knows a user’s password to perform a DoS attack!

"45flavors"
"46flavors"
"47flavors"
"48flavors"

Easy to guess honeywords!

Adversary can guess passwords simulating a DoS attack

Global password reset!

Inadequately sensitive

Overly sensitive
Multiple systems

Attack multiple systems against users that use the same password

**Intersection attack**

<table>
<thead>
<tr>
<th>Organisations A file F</th>
<th>Organisations B file F</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat93</td>
<td>cat93</td>
</tr>
<tr>
<td>cat54</td>
<td>cat74</td>
</tr>
<tr>
<td>cat22</td>
<td>cat28</td>
</tr>
<tr>
<td>cat42</td>
<td>cat62</td>
</tr>
</tbody>
</table>

Their intersection == cat93
That's user's password!

**Suggestion: take-a-tail**

| cat93 | cat15 | cat54 | cat74 | cat22 | cat28 | cat42 | cat62 |

Same head but different tail!

**Sweetword-submission attack**

Even if the adversary only has organisations A file F

| cat93 | cat54 | cat22 | cat42 |

Organisations B submit
No honeywords hit
RELATED WORK
**Related Work**

### Password strength
- **basic8** -> 1 billion guesses 40.3% cracked
- **MD5** -> 3 billion guesses/sec on GPUs
- The majority of passwords has around 20 **bits of entropy** against optimal attacker
  - 1 million guesses on average are enough
  - based on 70 million Yahoo! users
- **Bonneau and Preibusch** advice on:
  - password management
  - account lockout policies
  - update and recovery

### Password strengthening
- **take-a-tail** -> password strengthening
- **System generates** random characters until user obtains a memorable password
- e.g. **user’s suggestion** = ‘ilovecats’
- **system-generated** passwords:
  - ‘ilovecats523’
  - ‘ilovecats847’
  - ‘ilovecats196’
Password storage and verification

- **Splitting** password related secrets
  - distributed cryptography
- Preferable to honeywords
  - **require** big system and client **changes**
- **Honeywords** are a **stepping stone** to such approaches

Decoys

- Use of decoy resources is an old practice to detect security breaches!
- honeypots
- "Honeytokens" bogus credentials e.g.
  - fake credit card numbers
- Fabricated/decoy files
Conclusion
Conclusion

- Someone who has stolen a password file can \textbf{brute-force} to search for passwords.
- By using \textbf{honeywords} adversary does not have the confidence that he can login without being \textbf{detected}.
- Despite their benefits over common methods honeywords \textit{aren't a wholly satisfactory approach} to user authentications.
- \textbf{Simple}-to-deploy and a powerful new line of defense.
References

https://en.wikipedia.org/wiki/Bloom_filter
https://en.wikipedia.org/wiki/List_of_the_most_common_passwords
https://www.ece.unb.ca/tervo/ece4253/isbn.shtml