Making Password Checking Systems Better

Tom Ristenpart

Covering joint work with:
Anish Athayle, Devdatta Akawhe, Joseph Bonneau, Rahul Chatterjee,
Adam Everspaugh, Ari Juels, Sam Scott
Password checking systems

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tom</td>
<td>password1</td>
</tr>
<tr>
<td>alice</td>
<td>123456</td>
</tr>
<tr>
<td>bob</td>
<td>p@ssword!</td>
</tr>
</tbody>
</table>

(plus hundreds of millions more)

Allow login if:

- Attack detection mechanisms don’t flag request
- Password matches
- Sometimes: second factor succeeds
Problems w/ password checking systems

People often enter wrong password:
- Typos
- Memory errors

Passwords databases must be protected:
- Server compromise
- Exfiltration attacks (e.g., SQL injection)

Widespread practice:
- Apply hashing w/ salts
- Hope slows down attacks enough
Today’s talk

**Pythia**: moving beyond “hash & hope”

Harden hashes with off-system secret key using *partially oblivious pseudorandom function* protocol

[Chatterjee, Athayle, Akawhe, Juels, R. – Oakland 2016]

**Typo-tolerant password checking**

In-depth study of typos in user-chosen passwords

Show how to allow typos without harming security

[Chatterjee, Athayle, Akawhe, Juels, R. – Oakland 2016]
Password checking systems

Websites should *never* store passwords directly, they should be (at least) hashed with a salt (also stored)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tom</td>
<td>salt₁, Hₖ(password₁,salt₁)</td>
</tr>
<tr>
<td>alice</td>
<td>salt₂, Hₖ(123456,salt₂)</td>
</tr>
<tr>
<td>bob</td>
<td>salt₃, Hₖ(p@ssword!,salt₃)</td>
</tr>
</tbody>
</table>

Cryptographic hash function $H$ ($H = \text{SHA-256, SHA-512, etc.}$)

Common choice is $c = 10,000$

Better: scrypt, argon2

UNIX password hashing scheme, PKCS #5

Formal analyses: [Wagner, Goldberg 2000] [Bellare, R., Tessaro 2012]
## Password database compromises

<table>
<thead>
<tr>
<th>Year</th>
<th># Stolen</th>
<th>% Recovered</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>32.6 million</td>
<td>100%</td>
<td>plaintext (!)</td>
</tr>
<tr>
<td>2012</td>
<td>117 million</td>
<td>90%</td>
<td>Unsalted SHA-1</td>
</tr>
<tr>
<td>2013</td>
<td>36 million</td>
<td>??</td>
<td>ECB encryption</td>
</tr>
<tr>
<td>2015</td>
<td>36 million</td>
<td>33%</td>
<td>Salted bcrypt + MD5</td>
</tr>
</tbody>
</table>
(1) Password protections often implemented incorrectly in practice

(2) Even in best case, hashing slows down but does not prevent offline brute-force cracking
Facebook password onion

$cur = 'password'
$cur = md5($cur)
$salt = randbytes(20)
$cur = hmac_sha1($cur, $salt)
$cur = remote_hmac_sha256($cur, $secret)
$cur = scrypt($cur, $salt)
$cur = hmac_sha256($cur, $salt)
Strengthening password hash storage

- tom, password1
- \( h = H^c(password1 \| salt) \)
- Store salt, f
- \( f = f' \)?
- \( f' = H^c(123456 \| salt) \)
- \( f' = HMAC(K, h') \)
- \( H^c(1234567 \| salt) \)
- \( H^c(12345 \| salt) \)
- \( \vdots \)
- Back-end crypto service

HMAC is pseudorandom function (PRF).

Must still perform online brute-force attack

Exfiltration doesn’t help
Strengthening password hash storage

Critical limitation: can’t rotate K to a new secret K’

- Idea 1: Version database and update as users log in
  - *But doesn’t update old hashes*

- Idea 2: Invalidate old hashes
  - *But requires password reset*

- Idea 3: Use secret-key encryption instead of PRF
  - *But requires sending keys to web server (or high bandwidth)*

The diagram shows a hash computation:
\[
h = H^c(\text{password1} || \text{salt})
\]
where `h` is the hash value stored, `f = HMAC(K, h)` is used for verification, and `K` is the secret key stored in a back-end crypto service.
The Pythia PRF Service

Blinding means service learns *nothing* about passwords

Blind $h$, pick user ID

Unblind PRF output $f$

Store user ID, salt, $f$

User ID reveals fine-grained query patterns to service.
Compromise detection & rate limiting

Cryptographically erases $f$: Useless to attacker in the future

Server learns nothing about $K$ or $K'$

Combine token and $f$ to generate $f' = F(K', h)$

# Diagram

- **tom, password1** → **user id, blinded $h$** → **Blinded PRF output $f$**
- $h = H^c(password1 || salt)$
- Blind $h$, pick user ID
- Unblind PRF output $f$
- Store user ID, salt, $f$
- **Token($K' \rightarrow K'$)**
- **Back-end crypto service**
- **Cryptographically erases $f$: Useless to attacker in the future**
- **Server learns nothing about $K$ or $K'$**
New crypto: partially-oblivious PRF

Groups $G_1$, $G_2$, $G_T$ w/ bilinear pairing $e : G_1 \times G_2 \rightarrow G_T$  
\[ e(a^x, b^y) = c^{xy} \]

Choose random $r$  
\[ f = y^{1/r} \]

Store user ID, salt, $f$  

- Pairing cryptographically binds user id with password hash
- Can add verifiability (proof that PRF properly applied)
- Key rotation straightforward:  
  \[ \text{Token}(K \rightarrow K') = K' / K \]
- Interesting formal security analysis (see paper)
The Pythia PRF Service

- Queries are fast despite pairings
  - PRF query: 11.8 ms (LAN) 96 ms (WAN)
- Parallelizable password onions
  - $H^c$ and PRF query made in parallel (hides latency)
- Multi-tenant (theoretically: scales to 100 million login servers)
- Easy to deploy
Today’s talk

**Pythia**: moving beyond “hash & hope”

Harden hashes with off-system secret key using *partially oblivious pseudorandom function* protocol

[Everspaugh, Chatterjee, Scott, Juels, R. – USENIX Security 2015]

**Typo-tolerant password checking**

In-depth study of typos in user-chosen passwords
Show how to allow typos without harming security

[Chatterjee, Athayle, Akawhe, Juels, R. – Oakland 2016]
People often enter wrong password:
- Typos
- Memory errors

Passwords databases must be protected:
- Server compromise
- Exfiltration attacks (e.g., SQL injection)

Widespread practice:
- Apply hashing w/ salts
- Hope slows down attacks enough
Back to our big picture

People often enter wrong password:
- Typos
- Memory errors

Users have hard time remembering (complex) passwords
[Ur et al. 2012] [Shay et al. 2012] [Mazurek et al. 2013] [Shay et al. 2014]
[Bonneau, Schechter 2014]

Passwords can be difficult to enter without error (typo)
[Keith et al. 2007, 2009] [Shay et al. 2012]

Suggestions for error-correcting passphrases
[Bard 2007] [Jakobsson, Akavipat 2012] [Shay et al. 2012]
Facebook passwords are not case sensitive (update)

If you have characters in your Facebook password, there's a second password that you can log in to the social network with.

By Emil Protalinski for Friending Facebook | September 13, 2011 -- 12:26 GMT (05:26 PDT) | Topic: Security

password1       Password1       PASSWORD1

Typo-tolerant password checking:
Allow registered password or some typos of it
We focus on relaxed checkers

Apply typo corrector functions to incorrect submitted password:

- Apply caps lock corrector
- Apply first case flip corrector

Slow to compute $G_K$

Can we find small but useful set of typo correctors?

Works with existing password hardening schemes
No change in what is stored
Mechanical Turk transcription study

100,000+ passwords typed by 4,300 workers

% OF TYPOS

- Capslock 11%
- Flip first letter case 4.5%
- Add character at end 4.6%
- Other 78.8%

Top 3 account for 20% of typos

CAPS LOCK PREVENTING LOGIN SINCE 1980
Impact of Top 3 typos in real world

Instrumented production login of Dropbox to quantify typos

**NOTE:** We did not admit login based on relaxed checker

24 hour period:

- **3% of all users** failed to login because one of top 3 typos
- **20%** of users who made a typo would have saved at least 1 minute in logging into Dropbox if top 3 typos are corrected.

Allowing typos in password will add several person-months of login time every day.
Typo-tolerance would significantly improve usability of password-based login

Can it be secure?
Threat #1: Server compromise

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tom</td>
<td>$G_K(\text{password1})$</td>
</tr>
<tr>
<td>alice</td>
<td>$G_K(123456)$</td>
</tr>
<tr>
<td>bob</td>
<td>$G_K(\text{p@ssword!})$</td>
</tr>
</tbody>
</table>

No change to password DB

No change in security after compromise
Threat #2: Remote guessing attacks

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tom</td>
<td>$G_K(\text{password1})$</td>
</tr>
<tr>
<td>alice</td>
<td>$G_K(123456)$</td>
</tr>
<tr>
<td>bob</td>
<td>$G_K(p@ssword!)$</td>
</tr>
</tbody>
</table>

Apply caps lock corrector
Apply first case flip corrector
Apply extra char corrector

$G_K(\text{password})$
$G_K(\text{PASSWORD})$
$G_K(\text{Password})$
$G_K(\text{passwor})$
Threat #2: Remote guessing attacks

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tom</td>
<td>$G_K$(password1)</td>
</tr>
<tr>
<td>alice</td>
<td>$G_K$(123456)</td>
</tr>
<tr>
<td>bob</td>
<td>$G_K$(p@ssword!)</td>
</tr>
</tbody>
</table>

Server locks account after q failed attempts (e.g., q=10)

Apply caps lock corrector $G_K$(iloveyou)
Apply first case flip corrector $G_K$(ILOVEYOU)
Apply extra char corrector $G_K$(Iloveyou)

Up to 4 passwords checked at cost of 1 query

$\Rightarrow$

Attack success increases by 4x
Attack simulation using password leaks

Adversary knows:
   Distribution of passwords, and the set of correctors ()

**Exact checking**
Query most probable q passwords

**Typo-tolerant checking**
Query q passwords that maximizes success NP-complete problem.
Compute using greedy approximation

![Success probability graph](image)
Don’t check a correction if the resulting password is too popular.

Free Corrections Theorem:
For any password distribution, set of correctors, and query budget $q$, there exists a typo-tolerant checking scheme with no loss in security.
Security-sensitive typo tolerance

Assume distribution over passwords and order them in decreasing probability:

\[ \text{pw}_1 \text{ pw}_2 \ldots \text{ pw}_q \text{ pw}_{q+1} \text{ pw}_{q+2} \text{ pw}_{q+3} \ldots \]

Construction:
For any password, check as many typos as one can while ensuring correctness and that \( \sum \text{pw corrected} \Pr[\text{pw}] \leq \Pr[\text{pw}_q] \)

Ensures optimal adversarial strategy is to query \( \text{pw}_1, \ldots, \text{pw}_q \) against typo-tolerant checker. Same as for strict checker
Checkers w/ heuristic filtering

Use password leak rockyou to estimate distribution

Success probability (%)

- Exact checking
- Typo-tolerant checking
- Typo-tolerant checking w/ filtering

q = 10

phpbb

myspace

0.79 0.96 0.81
Typo-tolerance can enhance user experience without degrading security in practice
Today’s talk

**Pythia**: moving beyond “hash & hope”

Harden hashes with off-system secret key using *partially oblivious pseudorandom function* protocol

[Everspaugh, Chatterjee, Scott, Juels, R. – USENIX Security 2015]

**Typo-tolerant password checking**

In-depth study of typos in user-chosen passwords
Show how to allow typos without harming security

[Chatterjee, Athayle, Akawhe, Juels, R. – Oakland 2016]