Comment corriger efficacement les typos dans les mots de passe

Nikola K Blanchard

Comment corriger efficacement les typos dans les mots de passe

Nikola K. Blanchard, IRIF, Université Paris VII

Les mots de passe demeurant la principale méthode d'authentification en ligne, les progrès actuels se concentrent sur le lien entre utilisabilité et sécurité, et comment travailler sur la première améliore la deuxième. Dans une récente série de papiers, Chatterjee et al. ont introduit les premiers systèmes de mot-de-passe permettant de tolérer certaines typos, corrigeant jusqu'à 32% de celles-ci. Le système proposé ici corrige 57% des typos, (soit 91% des typos n'entraînant pas de risque réel), en ayant un coût négligeable côté serveur.

Mots-clefs : Mots de passe, Hachage, Sécurité accessible

1 Introduction

Malgré de nombreuses avancées en systèmes biométriques [Mem17], les mots de passe restent universels, et le seront sans doute durablement. Les risques liés à l'utilisation de ces derniers sont connus et analysés [MCTK10, Bon12], tout comme le fait que tout système de contrainte pousse les utilisateurs à être plus inventifs dans l'évasion desdites contraintes [Lip16, UNB+15]. Une procédure corrigeant automatiquement une erreur dans un mot de passe semble alors être une erreur affaiblissant encore plus un système déjà faillible. Cependant, des travaux récents ont montré qu'il n'en était rien [CAA+16, CWP+17], et qu'il était possible de construire un système corrigeant 32% des typos (ou fautes de frappe) sans perdre en sécurité. En effet, vu les méthodes d'attaque en ligne (qui ne passent pas par la force brute mais par des dictionnaires) et la distance entre les mots de passe courants, certaines erreurs peuvent être corrigées sans augmenter la probabilité de succès d'une attaque optimisée. Cela a deux avantages : améliorer l'utilisabilité (le taux de fautes de frappe dans les essais de connections à Dropbox ayant été estimé à 3% dans [CAA+16]) et, ce faisant, rendre les mots de passe longs plus faciles à taper sans erreur (et donc moins frustrant) améliorant donc la sécurité.

Hachés Vu la légitimité de systèmes tolérants les typos, la question est de savoir comment le faire, car, les mots de passe étant censés être hachés, vérifier si une typo est présente n'est pas directement faisable. Une méthode naïve consisterait à stocker sur le serveur (ou à envoyer) la liste de tous les hachés légitimes (entre une centaine et quelques milliers pour des mots de passe de taille moyenne). Le système présenté dans [CWP+17] stocke uniquement une demi-douzaine de hachés mais ne permet de conserver que les erreurs les plus fréquentes, et donc de corriger celles qui sont déjà stockées. Il est aussi plus coûteux, utilisant de la cryptographie asymétrique et plus de calculs côté serveur. Le système présenté ici cherche à stocker un nombre limité de hachés tout en corrigeant une plus large gamme d'erreurs.

Types de typos Les fautes de frappe peuvent être catégorisées, et les données fournies dans [CAA+16] permettent de voir que les fautes les plus fréquentes sont les suivantes : substitution d'un caractère par un autre (31%, dont plus de la moité correspond à des touches voisines sur le clavier), inversion de majuscule sur tout le mot de passe (14%), insertion d'un unique caractère (14%), suppression d'un caractère unique (12%) substitution de deux caractères (12%), transposition de deux caractères adjacents – l'ordre étant donc inversé – (4%), majuscule inversée sur un seul caractère (4%). Corriger les substitutions, transpositions, insertions et majuscules permettrait alors d'accepter entre 65% et 74% des mots de passe avec typo.

2 Algorithmes

Le système proposé est un framework de plusieurs algorithmes basés sur le même principe, chacun permettant de corriger certains types d'erreurs. La méthode consiste à stocker non pas un haché unique mais une liste de hachés du mot de passe avec des caractères manquants, accompagné d'indicateurs permettant de vérifier si les caractères manquants envoyés au serveur sont les bons. Vérifier que le caractère manquant est correct est nécessaire pour prévenir les suppressions (et les risques liés). Le framework contient des algorithmes pour gérer les substitutions par des caractères adjacents, les transpositions et les insertions. À chaque fois, trois algorithmes sont nécessaires (celui pour la création du mot de passe, celui pour l'envoi des hachés par le client, et enfin celui pour la vérification des hachés par le serveur).

Data: Salts $S_0, S_1, ...S_5$, Password P of length n; Keyboard map M : Keys $\rightarrow [0;255]$; Hash function HASH; Pseudorandom number generator PRNG
Result: Main hash and lists of (hash / integer) and (hash / integer list) pairs
begin
  $H_0 \longleftarrow$HASH(Concat(S_0, P))
  for i from 1 to n **do**
    $PA_i \longleftarrow P \setminus P[i]$; $HA_i \longleftarrow$HASH(Concat(S_1, PA_i))
    Random_bits $\longleftarrow$ PRNG(Concat(S_2, P_i))
    $\pi_i \longleftarrow$ Brassard(Random_bits)
    $K_i \longleftarrow \pi_i(M(P[i]))$
  for i from 1 to $n-1$ **do**
    $PB_i \longleftarrow P \setminus \{P[i] \bigcup P[i+1]\}$
    $HB_i \longleftarrow$HASH(Concat(S_1, PB_i))
    for j from 1 to 4 **do**
      Random_bits[j] $\longleftarrow$ PRNG(Concat(S_2, P_i))
      $\pi_{i,j} \longleftarrow$Brassard(Random_bits[j])
    $KA_i \longleftarrow [\pi_{i,1}(M(P[i]))]$
    $KB_i \longleftarrow [\pi_{i,2}(M(P[i+1]))]$
    $KC_i \longleftarrow [\pi_{i,3}(M(P[i]))]$
    $KD_i \longleftarrow [\pi_{i,4}(M(P[i+1]))]$
  return
    $(H_0, (HA_i, K_i)_{1 \leq i \leq n}, (HB_i, KA_i, KB_i, KC_i, KD_i)_{1 \leq i \leq n-1})$

Algorithm 1: Algorithme de création

L'algorithme ci-contre combine les idées de plusieurs algorithmes pour gérer ces trois types d'erreurs. Il commence par calculer n hachés différents, où à chaque fois un unique caractère est retiré avant de calculer le haché. Ensuite, une permutation aléatoire est calculée en utilisant le reste du mot de passe comme graine, et l'image du numéro du caractère manquant est envoyée. Cela suffirait pour corriger les substitutions, mais pour corriger les transpositions, il faut aussi tester les couples de caractères adjacents. Les hachés du mot de passe avec deux caractères adjacents retirés sont aussi calculés et stockés, avec les images des caractères manquant, calculées avec quatre permutations distinctes pour empêcher un adversaire de se rendre compte que deux caractères adjacents sont identiques. Dans l'algorithme d'envoi, les hachés privés d'un caractère sont calculés, et les images par les différentes permutations des deux caractères restants ainsi que des caractères proches sur le clavier sont envoyées.

Data: Salts S_0, S_1, S_2, Password P of length n
Keyboard map M
Hash function HASH, Pseudorandom number generator PRNG
Result: Set of n (hash / integer list) pairs
begin
  $H_0 \longleftarrow$HASH(Concatenate$((S_0, P))$
  for i from 1 to $n-1$ **do**
    $P_i \longleftarrow P \setminus \{P[i] \bigcup P[i+1]\}$
    $H_i \longleftarrow$HASH$(S_1 + P_i)$
    for j from 1 to 4 **do**
      Random_bits
        $\longleftarrow$PRNG(Concatenate(S_{j+2}, P_i))
      $\pi_{i,j} \longleftarrow$ Brassard(Random_bits)
    $LA_i \longleftarrow [\pi_{i,1}(M(i))]$
    LA_i.append$(\pi_{i,1}(M(SHIFT(P[i]))))$
    foreach $a \in$ Neighbours$(P[i])$ **do**
      LA_i.append$(\pi_{i,1}(M(a))$
    $LB_i \longleftarrow [\pi_{i,2}(M(i+1))]$
    LB_i.append$(\pi_{i,2}(M(SHIFT(P[i+1]))))$
    foreach $a \in$ Neighbours$(P[i+1])$ **do**
      LB_i.append$(\pi_{i,2}(M(a))$
    $LC_i \longleftarrow [\pi_{i,3}(M(P[i+1]))]$
    $LD_i \longleftarrow [\pi_{i,4}(M(P[i]))]$
  return $(H_0, (H_i, LA_i, LB_i, LC_i, LD_i)_{1 \leq i \leq n-1})$

Algorithm 2: Algorithme d'envoi

Data: Length n, Original hash H_0, Original list (HA_i, K_i)
Original list $(HB_i, KA_i, KB_i, KC_i, KD_i)$
Received hash H'_0 and list $(H'_i, LA_i, LB_i, LC_i, LD_i)$
Result: ACCEPT if and only if the password has at most one acceptable typo.
begin
  if $H_0 = H'_0$ **then**
  else
    return ACCEPT
  for i from 1 to $n-1$ **do**
    if $HB_i = H'_i$ **then**
      for j from 1 to $|LA_i|$ **do**
        if $(LA_i[j] = KA_i$ AND $LB_i[1] = KB_i)$ OR $(LB_i[j] = KB_i$ AND $LA_i[1] = KA_i)$ **then**
          return ACCEPT
      if $LC_i[1] = KC_i$ AND $LD_i[1] = KD_i$ **then**
        return ACCEPT
    else
      if $HA_i = H'_i$ AND $LB_i[2] = KB_i$ **then**
        return ACCEPT
  if $HA_n = H'_n$ AND $LB_n[2] = KB_n$ **then**
    return ACCEPT
  return REJECT

Algorithm 3: Algorithme de vérification

Quand le serveur veut vérifier que le client a bien le bon mot de passe, il commence par comparer les deux listes de hachés pour voir si un des hachés au moins est correct. Si c'est un haché de deuxième catégorie (avec deux caractères enlevés), il faut vérifier qu'on est dans un des cas suivants. Ou alors l'image d'un caractère manquant stockée sur le serveur est la même que celle du premier caractère envoyée par l'une des deux premières permutations, et l'image du deuxième caractère est dans la liste des voisins par l'autre permutation (correspondant à une substitution par un voisin), ou alors les images correspondent aux images par les troisièmes et quatrièmes permutations (correspondant à une transposition). Si c'est un haché de première catégorie, on a une insertion, et il suffit de vérifier que l'image du premier caractère manquant correspond bien à l'image stockée sur le serveur.

3 Performances

Quelques points supplémentaires sur l'algorithme doivent d'abord être mentionnés :

— Le fait d'envoyer les images des caractères voisins à la place de les stocker a un avantage : changer de clavier ne change en rien la capacité à corriger les erreurs.

— La gestion de la touche majuscule restée active se fait par l'envoi d'un unique haché supplémentaire (déjà utilisée chez Facebook entre autres).

— L'algorithme de permutation utilisé est celui de Brassard, qui permet de construire une permutation aléatoire de façon paresseuse sans avoir besoin de nombreux bits aléatoires [BK88].

Les différents algorithmes proposés (dans la version complète anonymisée sur `https://tinyurl.com/y9e83zve`) ont des performances variables résumées dans le tableau suivant pour des mots de passe de longueur $n \leq 16$ caractères (les modes conservateurs et tolérants dépendent de la gestion des insertions).

Algorithme	Substitution	Transposition	Insertion	Complet
Calcul				
Permutations	n	$4n-4$	$4n-4$	$\max(4n-4,60)$
Hachés	$n+1$	n	n	$\max(n+1,17)$
Entiers	$n \times k$	$(n-1) \times 4k$	$(n-1) \times 4k$	$\max(4(n-1)k,60k)$
Stockage				
Hachés	$n+1$	n	$2n$	$\max(2n+1,33)$
Entiers	n	$4n$	$5n$	$\max(5n,80)$
Typos Gérées				
Conservateur	24.2 %	28.4 %	34.5 %	50.2 %
Tolérant	24.2 %	28.4 %	42.2 %	57.7 %

4 Sécurité

Il s'agit maintenant d'estimer la résistance de l'algorithme à différentes attaques. Tout d'abord, on peut considérer la possibilité d'une attaque par force brute ou par dictionnaire. Dans ce cas, une analyse simple permet de montrer que, en bruteforce pure on gagne un facteur incompressible d'au plus 114, correspondant au nombre de variantes d'un mot de passe qui sont acceptées. Cependant, en essayant des attaques par dictionnaire, on peut vite se rendre compte que le gain de temps est beaucoup plus faible. En supprimant deux caractères des mots de passe de plus de 10 caractères de la base de données RockYou [KKM+12], voilà le nombre de mots de passe à tester pour arriver à avoir 50% de tous les mots de passe restants :

Position des caractères supprimés	aucun	0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9
Mots de passe uniques (millions)	4.40	4.26	4.33	4.29	4.29	4.28	4.26	4.22	4.12	3.96
Proportion pour atteindre 50%	33.1	29.9	31.4	30.6	30.7	30.4	30.0	29.0	26.7	23.0
Facteur d'accélération	1	1.11	1.05	1.08	1.08	1.09	1.10	1.14	1.24	1.44

On peut donc voir que, vu la relativement grande variété de mots de passe (même dans cette base de donnée dont les mots de passe sont généralement considérés de mauvaise qualité), supprimer deux caractères ne suffit pas à rendre une attaque par force brute vraiment plus facile. Cela rejoint les résultats de [CWP+17], car empêcher la suppression et limiter la substitution rend le facteur d'accélération très proche de 1.

Sans accès au serveur, les attaques par force brute et par dictionnaire sont les seules où changer la méthode de hachage pour autoriser les typos peut affecter le niveau de sécurité. Si l'adversaire a accès au serveur cependant, l'utilisation de ces algorithmes a un coup réel, même en utilisant de bonnes pratiques (comme un sel différent à chaque mot de passe pour éviter les tables arc-en-ciel). Cette vulnérabilité est liée au fait que retrouver un mot de passe à partir d'un haché est extrêmement difficile et, avec de bons algorithmes, n'est pas rendu plus beaucoup plus facile par la présence de plusieurs hachés légèrement différents. Un adversaire ne cherche donc généralement pas à retrouver l'original à partir du haché, mais plutôt à tester de très nombreux mots de passe candidats en hachant à chaque fois pour essayer de tomber sur un haché connu. Pour empêcher cela, la méthode standard est de ralentir le hachage – les machines modernes pouvant calculer de dizaine de milliards de hachés par secondes, parcourant ainsi facilement un dictionnaire entier pour chaque utilisateur – avec des algorithmes comme Argon2 ou PBKDF2. En ralentissant le hachage pour que chaque mot de passe prenne 1ms à hacher, on peut se prémunir de ces attaques. Ici, une vulnérabilité est que ce temps passé à hacher est forcément partagé entre les différents hachés, divisant ainsi le coût de cette attaque par le nombre de hachés, égal à la longueur du mot de passe plus 1. Cela reste limité, et 15ms passées à calculer des hachés suffisent à atteindre un niveau de sécurité suffisant pour l'utilisateur moyen.

Le système présenté ici permet de corriger jusqu'à 57% des typos (soit 91% des typos ne présentant pas de dangers) sans vrai calcul du côté serveur et pouvant être utilisé avec une évaluation paresseuse, ne comparant les erreurs que quand c'est nécessaire. Il a aussi un coût de communication négligeable, l'ensemble des données pour les hachés et les numéros envoyés prenant au total 964 octets, bien en dessous de la moyenne pour un paquet IPv6. Il a un impact négligeable sur la sécurité, tant contre des adversaires essayant de déchiffrer les mots de passe que contre les adversaires essayant de s'introduire dans un compte. Enfin, dans sa version complète, c'est un framework facilement adaptable pour gérer de multiples types d'erreurs en autorisant uniquement certaines typos.

Références

[BK88] Gilles Brassard and Sampath Kannan. The generation of random permutations on the fly. *Inf. Process. Lett.*, 28(4) :207–212, July 1988.

[Bon12] J. Bonneau. The science of guessing : Analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, pages 538–552, 5 2012.

[CAA+16] Rahul Chatterjee, Anish Athayle, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. password typos and how to correct them securely. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 799–818. IEEE, 2016.

[CWP+17] Rahul Chatterjee, Joanne Woodage, Yuval Pnueli, Anusha Chowdhury, and Thomas Ristenpart. The typtop system : Personalized typo-tolerant password checking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 329–346, New York, NY, USA, 2017. ACM.

[KKM+12] Patrick Gage Kelley, Saranga Komanduri, Michelle L. Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lopez. Guess again (and again and again) : Measuring password strength by simulating password-cracking algorithms. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 523–537. IEEE, 2012.

[Lip16] Peter Lipa. The security risks of using "forgot my password" to manage passwords, 2016. Accessed : 2017-12-18.

[MCTK10] W. Ma, J. Campbell, D. Tran, and D. Kleeman. Password entropy and password quality. In *2010 Fourth International Conference on Network and System Security*, pages 583–587, 9 2010.

[Mem17] Nasir Memon. How biometric authentication poses new challenges to our security and privacy [in the spotlight]. *IEEE Signal Processing Magazine*, 34(4) :196–194, 2017.

[UNB+15] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie F. Cranor. I added '!'at the end to make it secure" : Observing password creation in the lab. In *Proc. SOUPS*, 2015.