

Topic 2.4

Password hashing

CO 487/687 • Applied Cryptography

Douglas Stebila

UNIVERSITY OF
WATERLOO



User authentication

- Authenticators can be categorised as:
 - **Knowledge-Based** (Something you know)
 - **Object-Based** (Something you have)
 - **ID-Based** (Something you are)
 - **Location-based** (Somewhere you are)
- **Multi-factor authentication** uses combinations from multiple different categories of authenticators

Passwords

- Passwords are human-memorizable strings that are used for authentication

Common attacks against passwords

- Attacker steals a password from a user (via malware, breaking kneecaps, ...)
- Attacker guesses a user's password
 - Through online guessing
- Attacker steals a password database from a server
 - Then uses offline computation
- Hard-coded passwords

Security recommendations for passwords

- Use a ‘strong’ password
 - Aspects include minimum length, character set, prohibiting use of identifiers or known associated items as passwords, limitation on length of time before change required
- Store password securely
 - Not on a post-it note on your monitor (?)
- Don’t share password with other entities
 - Colleagues, friends, family, etc.
- Don’t use same password for multiple systems
 - Different unrelated passwords for work/study, online banking, social media, etc.

RockYou.com password breach

- RockYou.com, a social media gaming site, had their password database compromised in 2009. Passwords were stored in plaintext.
- First large-scale password breach with publicly analyzed datasets
- # of accounts: 32.6 million
- # of different passwords: 14.3 million

RockYou.com password statistics

- About 30% of passwords length less than or equal to six characters.
- Nearly 50% of users used names, slang words, dictionary words or trivial passwords (consecutive digits, adjacent keyboard keys, and so on).
- Entropy of password set: 21.1 bits

Top 10 passwords:

1. 123456
2. 12345
3. 123456789
4. password
5. iloveyou
6. princess
7. 1234567
8. rockyou
9. 12345678
10. abc123

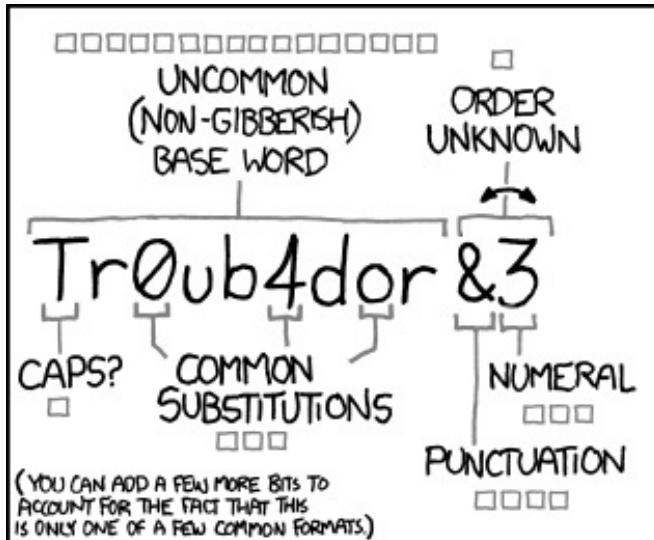
RockYou.com password statistics

- The top __ passwords covered __% of user accounts:

–	1	0.9%
–	5	1.7%
–	10	2.1%
–	100	4.6%
–	1000	11.3%
–	10000	22.3%

Computer-generated reusable passwords

- Computer generated passwords avoid the problem of users choosing weak passwords
- But have another security problem:
 - Passwords consisting of random characters difficult for users to remember, so they may write them down.
- Various mechanisms for generating human-friendly passwords:
 - Syllabic word-like: FIPS PUB 181
<http://csrc.nist.gov/publications/fips/fips181/fips181.pdf>
 - Sequences of words:
 - Diceware: <http://world.std.com/~reinhold/diceware.html>
 - xkcd: <http://correcthorsebatterystaple.net>, <https://xkpasswd.net/s/>



~28 BITS OF ENTROPY

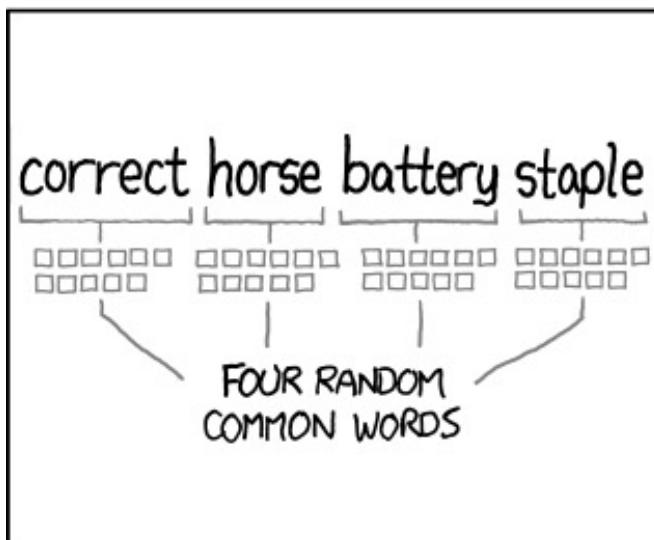
$2^{28} = 3 \text{ DAYS AT } 1000 \text{ GUESSES/SEC}$

(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)

DIFFICULTY TO GUESS:
EASY

WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE 0s WAS A ZERO?
AND THERE WAS SOME SYMBOL...

DIFFICULTY TO REMEMBER:
HARD



~44 BITS OF ENTROPY

$2^{44} = 550 \text{ YEARS AT } 1000 \text{ GUESSES/SEC}$

DIFFICULTY TO GUESS:
HARD

THAT'S A BATTERY STAPLE.
CORRECT!

DIFFICULTY TO REMEMBER:
YOU'VE ALREADY MEMORIZED IT

THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Entropy: a (somewhat okay) measure of password strength

- **Entropy** measures the uncertainty in values generated from a random process
- Think of passwords being generated from a random process with a certain distribution
- Predicts the number of guesses we have to make to learn the password

Not ideal measure of password guessing difficulty, but reasonable good (see http://jbonneau.com/doc/2012-jbonneau-phd_thesis.pdf for detailed analysis)

Entropy: a (somewhat okay) measure of password strength

- Suppose a process X generates n values x_1, \dots, x_n with probabilities p_1, \dots, p_n
- Formula for entropy of process X :

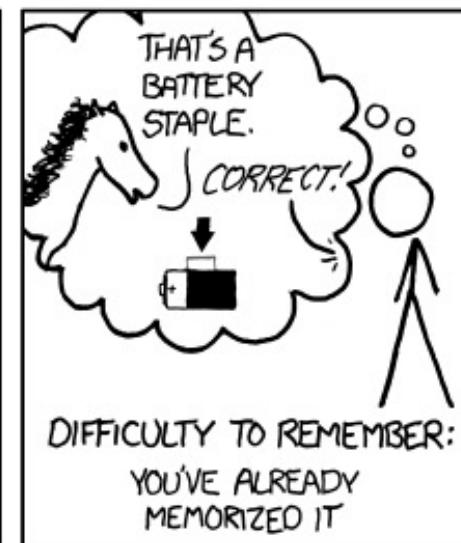
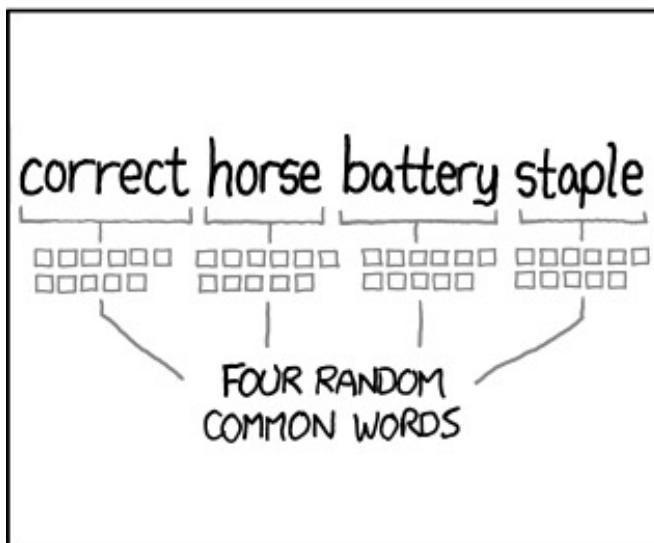
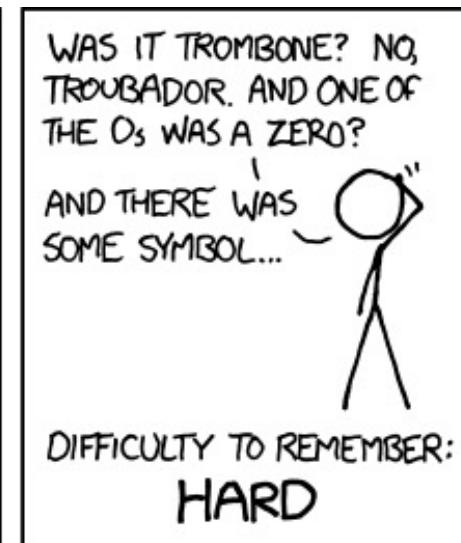
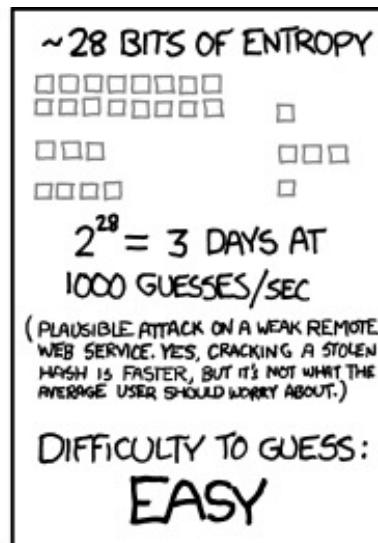
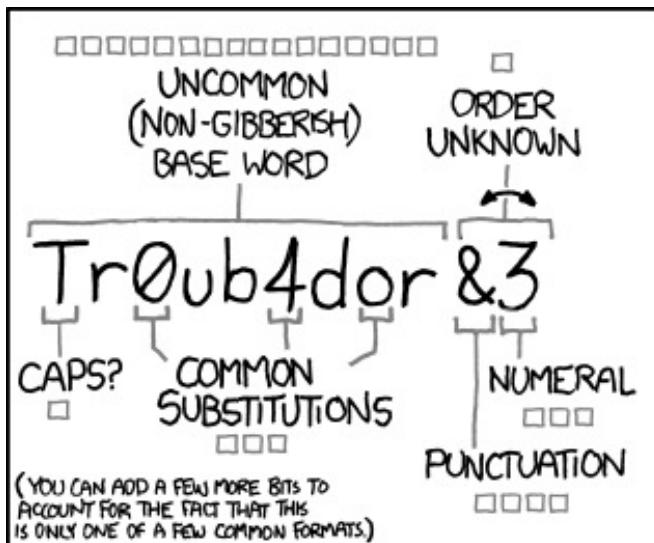
$$H(X) = - \sum_{i=1}^n p_i \log_2(p_i)$$

– Or alternatively:

$$H(X) = -p_1 \log_2(p_1) - p_2 \log_2(p_2) - \dots - p_n \log_2(p_n)$$

Entropy: a (somewhat okay) measure of password strength

- Simple way of thinking about it:
 - If a password is chosen uniformly at random from a set of size 2^n ,
 - then its entropy is n bits,
 - and we require around 2^{n-1} guesses on average to find it.



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD TO REMEMBER, BUT EASY FOR COMPUTERS TO CRACK.

RockYou.com: 21.1 bits of entropy

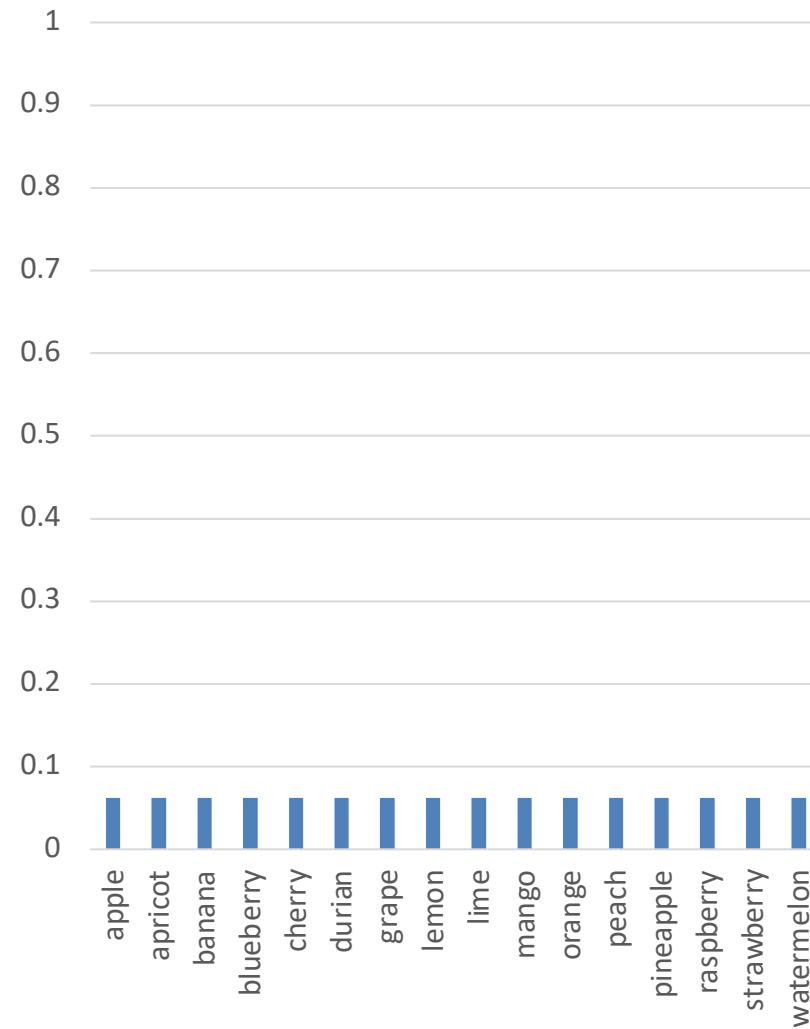
Example: calculating entropy

- Suppose we have a dictionary of 16 words.
- Scenario 1: Passwords generated uniformly at random from the dictionary
 - i.e., each password is equally likely
- Scenario 2: Passwords were NOT generated uniformly at random from the dictionary
 - i.e., some passwords more likely than others

Example: calculating entropy

Scenario 1: Equally likely passwords

Password (x_i)	Probability (p_i)
apple	1/16
apricot	1/16
banana	1/16
blueberry	1/16
cherry	1/16
durian	1/16
grape	1/16
lemon	1/16
lime	1/16
mango	1/16
orange	1/16
peach	1/16
pineapple	1/16
raspberry	1/16
strawberry	1/16
watermelon	1/16



Example: calculating entropy

Scenario 1: Equally likely passwords

Password (x_i)	Probability (p_i)
apple	1/16
apricot	1/16
banana	1/16
blueberry	1/16
cherry	1/16
durian	1/16
grape	1/16
lemon	1/16
lime	1/16
mango	1/16
orange	1/16
peach	1/16
pineapple	1/16
raspberry	1/16
strawberry	1/16
watermelon	1/16

$$\begin{aligned}
 H(X) &= - \sum_{i=1}^{16} p_i \log_2(p_i) \\
 &= - \sum_{i=1}^{16} \frac{1}{16} \log_2 \left(\frac{1}{16} \right) \\
 &= -16 \cdot \frac{1}{16} \log_2 \left(\frac{1}{16} \right) \\
 &= -1 \cdot \log_2 \left(\frac{1}{16} \right) \\
 &= -1 \cdot \log_2 (2^{-4}) \\
 &= 4
 \end{aligned}$$

Example: calculating entropy

Scenario 1: Equally likely passwords

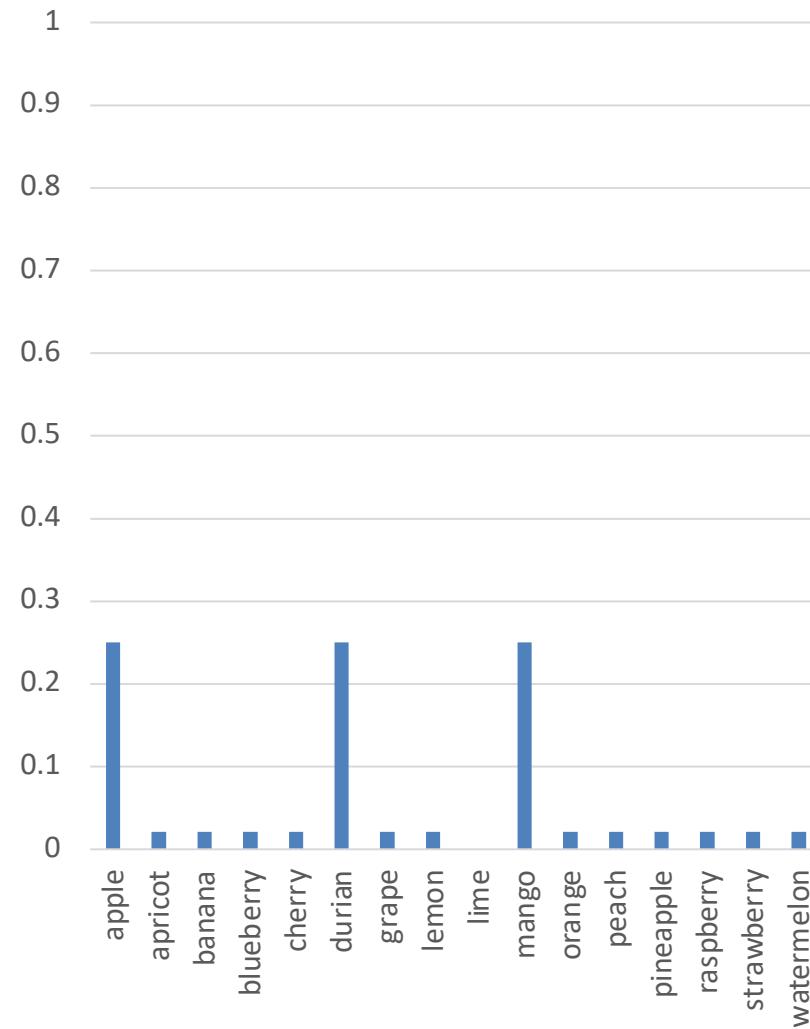
Password (x_i)	Probability (p_i)
apple	1/16
apricot	1/16
banana	1/16
blueberry	1/16
cherry	1/16
durian	1/16
grape	1/16
lemon	1/16
lime	1/16
mango	1/16
orange	1/16
peach	1/16
pineapple	1/16
raspberry	1/16
strawberry	1/16
watermelon	1/16

If you are trying to guess the password, you need to make about $2^{4-1} = \mathbf{8 \text{ guesses}}$ on average

Example: calculating entropy

Scenario 2: Non-uniform passwords

Password (x_i)	Probability (p_i)
apple	1/4
apricot	1/48
banana	1/48
blueberry	1/48
cherry	1/48
durian	1/4
grape	1/48
lemon	1/48
lime	0
mango	1/4
orange	1/48
peach	1/48
pineapple	1/48
raspberry	1/48
strawberry	1/48
watermelon	1/48



Example: calculating entropy

Scenario 2: Non-uniform passwords

Password (x_i)	Probability (p_i)
apple	1/4
apricot	1/48
banana	1/48
blueberry	1/48
cherry	1/48
durian	1/4
grape	1/48
lemon	1/48
lime	0
mango	1/4
orange	1/48
peach	1/48
pineapple	1/48
raspberry	1/48
strawberry	1/48
watermelon	1/48

$$\begin{aligned}
 H(X) &= - \sum_{i=1}^{16} p_i \log_2(p_i) \\
 &= -3 \cdot \frac{1}{4} \log_2\left(\frac{1}{4}\right) \\
 &\quad - 12 \cdot \frac{1}{48} \log_2\left(\frac{1}{48}\right) \\
 &\quad - 0 \\
 &\approx -\frac{3}{4} \log_2(2^{-2}) \\
 &\quad - \frac{12}{48} \log_2(2^{-5.59}) \\
 &= \frac{3}{4} \cdot 2 + \frac{12}{48} \cdot 5.59 \\
 &= 2.8975
 \end{aligned}$$

Example: calculating entropy

Scenario 2: Non-uniform passwords

Password (x_i)	Probability (p_i)
apple	1/4
apricot	1/48
banana	1/48
blueberry	1/48
cherry	1/48
durian	1/4
grape	1/48
lemon	1/48
lime	0
mango	1/4
orange	1/48
peach	1/48
pineapple	1/48
raspberry	1/48
strawberry	1/48
watermelon	1/48

If you are trying to guess the password, you need to make about $2^{2.8975-1} = 3$ guesses on average

Entropy

- If some words are more likely than others, there's less uncertainty
=> less entropy
=> easier to guess
- Entropy of passwords is a combination of length of password and randomness of each part of the password

The screenshot shows a web browser window displaying an Ars Technica article. The URL in the address bar is <https://arstechnica.com/security/2024/09/nist-proposes-barring-some-of-the-most-nonsensical-password-rules/>. The page features the Ars Technica logo and navigation links for 'SECTIONS', 'FORUM', 'SUBSCRIBE', and 'SIGN IN'. A prominent message 'YOUR PASSWORD HAS EXPIRED' with a skull icon is visible. The main title of the article is 'NIST proposes barring some of the most nonsensical password rules', followed by a subtitle 'Proposed guidelines aim to inject badly needed common sense into password hygiene.' and the author's name 'DAN GOODIN – SEP 25, 2024 6:39 PM'. Below the article, there is a sidebar with a '343' comment count and a text block about NIST's proposed password rules.

343

The National Institute of Standards and Technology (NIST), the federal body that sets technology standards for governmental agencies, standards organizations, and private companies, has proposed barring some of the most vexing and nonsensical password requirements. Chief among them: mandatory resets, required or restricted use of certain characters, and the use of security questions.

Choosing strong passwords and storing them safely is one of the most challenging parts of a good cybersecurity regimen. More challenging still is complying with password rules imposed by employers, federal agencies, and

<https://arstechnica.com/security/2024/09/nist-proposes-barring-some-of-the-most-nonsensical-password-rules/>

1. Verifiers and CSPs **SHALL** require passwords to be a minimum of eight characters in length and **SHOULD** require passwords to be a minimum of 15 characters in length.
2. Verifiers and CSPs **SHOULD** permit a maximum password length of at least 64 characters.
3. Verifiers and CSPs **SHOULD** accept all printing ASCII [[RFC20](#)] characters and the space character in passwords.
4. Verifiers and CSPs **SHOULD** accept Unicode [[ISO/ISC 10646](#)] characters in passwords. Each Unicode code point **SHALL** be counted as a single character when evaluating password length.
5. Verifiers and CSPs **SHALL NOT** impose other composition rules (e.g., requiring mixtures of different character types) for passwords.
6. Verifiers and CSPs **SHALL NOT** require users to change passwords periodically. However, verifiers **SHALL** force a change if there is evidence of compromise of the authenticator.
7. Verifiers and CSPs **SHALL NOT** permit the subscriber to store a hint that is accessible to an unauthenticated claimant.
8. Verifiers and CSPs **SHALL NOT** prompt subscribers to use knowledge-based authentication (KBA) (e.g., “What was the name of your first pet?”) or security questions when choosing passwords.
9. Verifiers **SHALL** verify the entire submitted password (i.e., not truncate it).

Password hashing

STORING PASSWORDS ON SERVERS

Login and registration, take 1

Registration

1. Store username and password in database

Login

1. User supplies username and purported password
2. Look up username and real password in database
3. Check if purported password = real password

'-- Have I Been Pwned: Check if you have been compromised in a data breach haveibeenpwned.com

'--have i been pwned?

Check if you have an account that has been compromised in a data breach

 Generate secure, unique passwords for every account [Learn more at 1Password.com](#)

Why 1Password?

408 pwned websites 8,506,873,299 pwned accounts 102,437 pastes 122,479,810 paste accounts

Largest breaches

	772,904,991 Collection #1 accounts
	763,117,241 Verifications.io accounts
	711,477,622 Onliner Spambot accounts
	593,427,119 Exploit.In accounts
	457,962,538 Anti Public Combo List accounts
	393,430,309 River City Media Spam List accounts
	359,420,698 MySpace accounts
	234,842,089 NetEase accounts
	164,611,595 LinkedIn accounts
	161,749,950 Dubsmash accounts

Recently added breaches

	988,230 StreetEasy accounts
	780,073 Sephora accounts
	23,165,793 Wanelo accounts
	15,453,048 Lumin PDF accounts
	4,606 KiwiFarms accounts
	396,533 Minehut accounts
	95,431 Void.to accounts
	36,395,491 Poshmark accounts
	89,388 Mastercard Priceless Specials accounts
	561,991 XKCD accounts

<https://haveibeenpwned.com/PwnedWebsites>

Storing passwords securely

- Security requirements for system files storing passwords:
 - C: Can non-administrators read the password database? What useful information is in there?
 - I: Can the password file be modified? Can unauthorised modification be detected?
 - A: Need to be available when required for verification
- Note: no non-repudiation if password is known to system (or to others outside the system)

Confidentiality of passwords

- **Storage** (on authentication server)
- **Transmission** (between client and server over network)
- **Use** (display on screen when being entered?)

Login and registration, take 2

Registration

1. Store username and an **encrypted version** of the password in database

Problem: if someone learns the key, they can decrypt the database and recover all the passwords.

Login

1. User supplies username and purported password
2. Look up username and **encrypted password** in database
3. **Decrypt the stored password to recover the real password**
4. Check if purported password = real password

Login and registration, take 3

Registration

1. Store username and an **irreversible transformation (hash)** of the password in database

Login

1. User supplies username and purported password
2. Look up username and **hash** in database
3. **Apply same irreversible transformation to the purported password**
4. Check if **hash** of purported password = **hash** of real password

Using password hashes for login

Benefits

- Compromise of the database doesn't reveal the user's password
- Almost no overhead for storage and login

Drawbacks

- Can't recover passwords for users who forget
- Attackers could create a table of password hashes to compare against database
- Can learn if two users use the same password (even if you don't know what it is)

Password hash cracking

- Suppose you learn that the hash of Alice's password is `3e2e95f5ad970eadfa7e17eaf73da97024aa5359`
 - Maybe by a database breach
- Goal: find Alice's password

Brute force attack

- Search through all possible passwords
- Possibly ordered by frequency based on known human-picked password distributions
- How big is a password space?
 - A-Z=26, a-z=26, 0-9=10
 - 8 character password
 - $62^8 = 2^{47.6}$ possible passwords
- How much can one computer do?
 - On a single computer, this would take around 1 year
 - < \$200 on Amazon
 - < \$50 on a botnet

Attacking using hash tables

- **Hash table:** A table containing hashes of many/all possible passwords
- Would allow an attacker with the password database to quickly find the user's password.
- More work to crack one password hash, but can reuse work ("precomputation") to crack many password

Attacking using hash tables

- Hash tables allow for instant cracking of a password hash
- But require a massive amount of storage
 - password set: 8 character passwords, $26+26+10=62$ characters
 - $62^8 = 2^{47.6}$ passwords
 - SHA-1 hash table would take 160 bits = 20 bytes per password
 - approx. $2^{52.4}$ bytes = 6 petabytes
- Can we find a time-memory trade-off where we can store less, but not increase time too much?

Precomputed hash tables

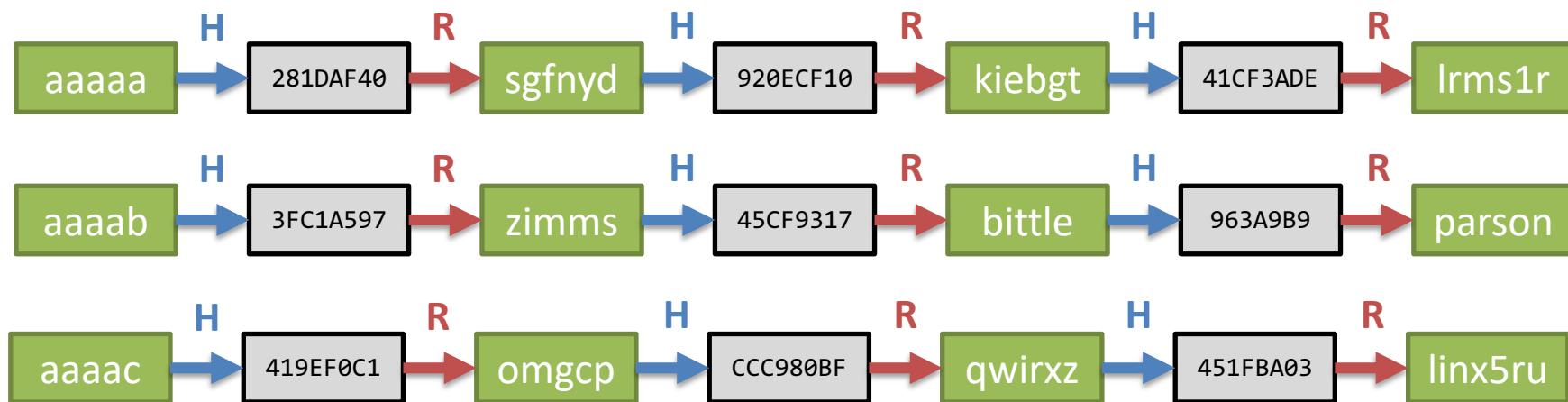
- Lots of precomputed hash tables available online from legitimate and dubious websites
 - <https://a.ndronic.us/pre-computed-hash-table-v-1-0/>
 - <https://hashes.org>
 - <https://www.google.com/>

Attacking using rainbow tables

- **Rainbow tables** are an example of a time-space tradeoff using hash chains.
- Ophcrack and RainbowCrack are examples of software that can crack passwords using rainbow tables.
- RainbowCrack example:
 - 1-8 character mixed-case alphanumeric password
 - 160GB rainbow table
 - time to crack 1 password using CPU: approx. 26 minutes
 - time to crack 1 password using GPU: approx. 103 seconds
 - success rate: 99.9%

Constructing a rainbow table

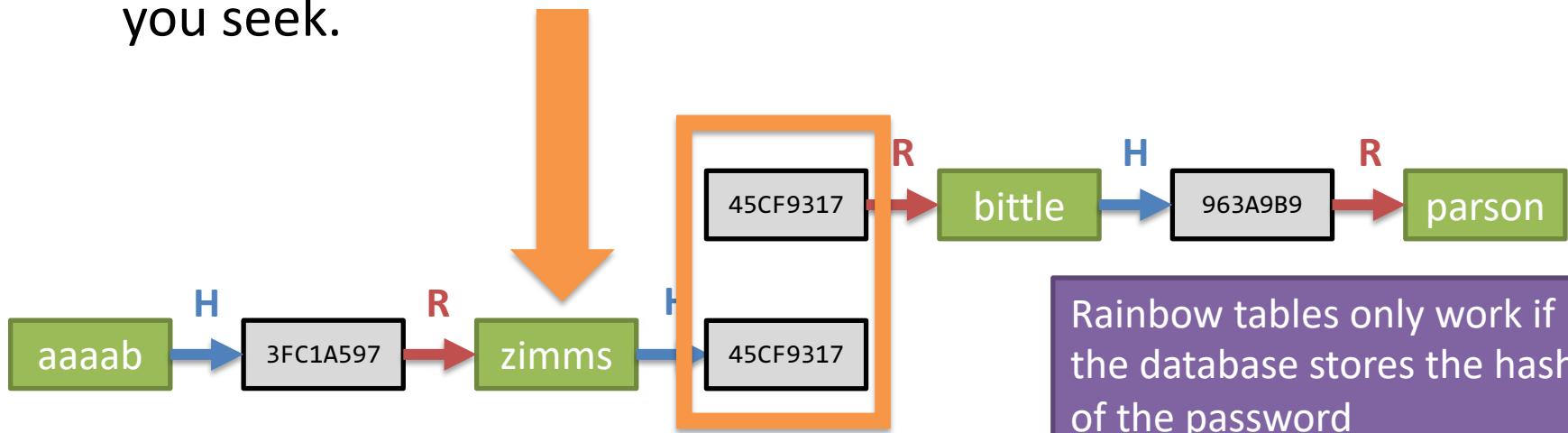
1. Pick a random password
2. Construct a hash chain (hash with H, map hash back to the password space with R)
3. Store the start and end of the chain
4. Repeat many times



Using a rainbow table

1. Given a hash **45CF9317**
2. Construct a hash chain from that hash (R, H, R, H, ...), each time checking to see if the value matches any stored tail.
3. Once tail is found, take the corresponding head, and construct a hash chain (H, R, H, R, ...) until you find your hash
4. The one immediately before is the password you seek.

Rainbow table	
Head	Tail
aaaaa	Irms1r
aaaab	parson
aaaac	linx5ru



Rainbow tables only work if the database stores the hash of the password $H(\text{password})$.

Login and registration, take 4

Registration

1. Pick a random ≥ 80 -bit salt
2. Store username, salt, and $H(\text{password}, \text{salt})$ in database where H is a cryptographic hash function

Login

1. User supplies username and purported password'
2. Look up username, salt, and hash in database
3. Check if $H(\text{password}', \text{salt}) = \text{stored hash}$

Benefits of salting

- Salting **protects against rainbow tables** since you would need a different table for each salt.
- Salting **makes brute-force attacks harder** because you can't reuse the work from one attack on another attack.

Password hardening

- You can slow down brute-force attacks even more by **hashing the password multiple times**.
- Instead of storing
 $H(\text{salt}, \text{password})$
store
 $H(H(H(\dots H(\text{salt}, \text{password}))))$
with 10000 hash function applications.
- My computer can apply SHA1 3190046 times per second
- So 10000 times only takes in 0.003 seconds
- Doesn't slow down login much.
- But it does slow down brute-force attacks by a factor of 10000.

Password hardening functions

- PBKDF2 (2000)
 - Widely used; fairly secure
- bcrypt
- scrypt
- Argon2 (2015)
 - Best available approach

Login and registration, take 5

Registration

1. Pick a random ≥ 80 -bit salt
2. Store username, salt, and $H(\text{password}, \text{salt})$ in database **where H is a password hardening function**

Login

1. User supplies username and purported password'
2. Look up username, salt, and hash in database
3. Check if $H(\text{password}', \text{salt}) = \text{stored hash}$

FACEBOOK PASSWORD HASHING

Case study

Facebook password hashing as of 2014



PASSWORDS 2014 Conference
December 8-10, 2014 - Trondheim, Norway

Yes, we call it “The Onion”

```
$cur = 'plaintext'  
$cur = md5($cur)  
$salt = randbytes(20)  
$cur = hmac_sha1($cur, $salt)  
$cur = cryptoservice::hmac($cur)  
    [= hmac_sha256($cur, $secret)]  
$cur = scrypt($cur, $salt)  
$cur = hmac_sha256($cur, $salt)
```



FRISC

NTNU – Trondheim
Norwegian University of
Science and Technology

Steps of the Facebook password onion

Operation	Comment
1. Hash using MD5	Easy to inject password dumps from breaches for proactive breach detection
2. Hash using HMAC-SHA1 using 160-bit salt	Defeats pre-computation attacks; 160-bit salt good when dealing with $2^{31}+$ users
3. Hash using HMAC-SHA256 on a remote server with a hidden key	Forces authentication checks to go through a centrally monitored server, hard to generate fake hashes
4. Hash using scrypt password hardening function (using 0.2% of front-end server CPU)	Slows down brute force attacks
5. Hash with HMAC-256	Shrinks the output of scrypt back to a smaller size

Additional reason for this approach

- Password hashing technology evolves over time
- Nesting "older" hashing approaches inside newer ones lets you upgrade technology without waiting for all users to log in so you can generate a new hash using the latest technology

PASSWORDS 2014 Conference
December 8-10, 2014 - Trondheim, Norway

Benefits

- Nesting hashes makes great sense “at scale”
 - ...rather than expecting 1.35 billion people to log-in just because you want to expire the old hashes

The screenshot shows a web browser window with the Ars Technica website open. The title of the article is "Meta pays the price for storing hundreds of millions of passwords in plaintext". Below the title, a sub-headline reads "Company failed to follow one of the most sacrosanct rules for password storage." The author's name, DAN GOODIN, and the publication date, SEP 27, 2024 1:53 PM, are visible. A sidebar on the left contains a comment count of 188. The main text discusses a \$101 million fine imposed by Irish officials for storing user passwords in plaintext. A secondary text block below provides more details about the disclosure and querying of the database.

ars Meta pays the price for storin +

arstechnica.com/security/2024/09/meta-slapped-with-101-million-fine-for-storing-passwords-in-plaintext/ ☆ 🔍 📁 📸 🗂️ 🌐 🌐 🌐

ars TECHNICA

GOT HASHES?

Meta pays the price for storing hundreds of millions of passwords in plaintext

Company failed to follow one of the most sacrosanct rules for password storage.

DAN GOODIN – SEP 27, 2024 1:53 PM

188

Officials in Ireland have fined Meta \$101 million for storing hundreds of millions of user passwords in plaintext and making them broadly available to company employees.

Meta disclosed the lapse in early 2019. The company said that apps for connecting to various Meta-owned social networks had logged user passwords in plaintext and stored them in a database that had been searched by roughly 2,000 company engineers, who collectively queried the stash more than 9 million times.

The screenshot shows a web browser window with the URL krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/. The page features a large header with the text "KrebsOnSecurity" and "In-depth security news and investigation". Below the header is a portrait of a man in a suit. The main title of the article is "Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years". The publication date is March 21, 2019, and there are 208 comments. The article content discusses Facebook's handling of user passwords.

Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years

March 21, 2019

208 Comments

Hundreds of millions of **Facebook** users had their account passwords stored in plain text and searchable by thousands of Facebook employees — in some cases going back to 2012, KrebsOnSecurity has learned. Facebook says an ongoing investigation has so far found no indication that employees have abused access to this data.

Facebook is probing a series of security failures in which employees built applications that logged unencrypted password data for Facebook users and stored it in plain text on internal company servers. That's according to a senior Facebook employee who is familiar with the investigation and who spoke on condition of anonymity because they were not authorized to speak to the press.

The Facebook source said the investigation so far indicates between 200 million and 600 million Facebook users may have had their account passwords stored in plain text and searchable by more than 20,000 Facebook employees. The source said Facebook is still trying to determine how many passwords were exposed and for how long, but so far the inquiry has uncovered archives.

Facebook Stored Hundreds of ...          

KrebsOnSecurity

In-depth security news and investigation



HOME ABOUT THE AUTHOR ADVERTISING/SPEAKING

Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years

The Facebook source said the investigation so far indicates between 200 million and 600 million Facebook users may have had their account passwords stored in plain text and searchable by more than 20,000 Facebook employees. The source said Facebook is still trying to determine how many passwords were exposed and for how long, but so far the inquiry has uncovered archives with plain text user passwords dating back to 2012.

My Facebook insider said access logs showed some 2,000 engineers or developers made approximately nine million internal queries for data elements that contained plain text user passwords.

Investigation and who spoke on condition of anonymity because they were not authorized to speak to the press.

The Facebook source said the investigation so far indicates between 200 million and 600 million Facebook users may have had their account passwords stored in plain text and searchable by more than 20,000 Facebook employees. The source said Facebook is still trying to determine how many passwords were exposed and for how long, but so far the inquiry has uncovered archives with plain text user passwords dating back to 2012.

Lessons learned

- Facebook password hashing had a good design that could evolve over time
- Mistakes in other parts of the system (logging application data) undermined security
- Not handling user data properly can lead to substantial consequences