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

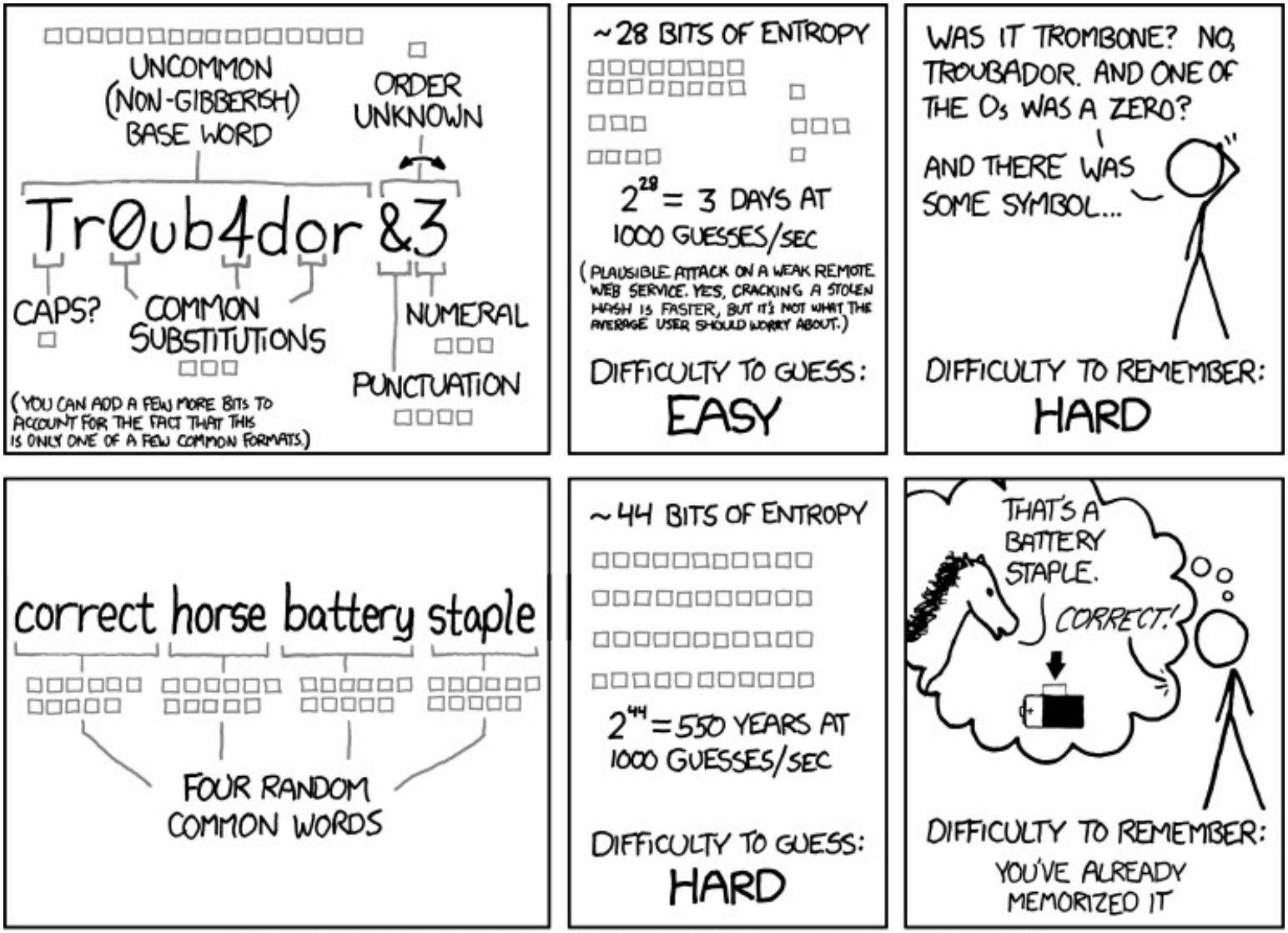
http://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf

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/



https://xkcd.com/936/

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 $x_1, \ldots, x_n$ with probabilities $p_1, \ldots, p_n$

- Formula for entropy of process *X*:

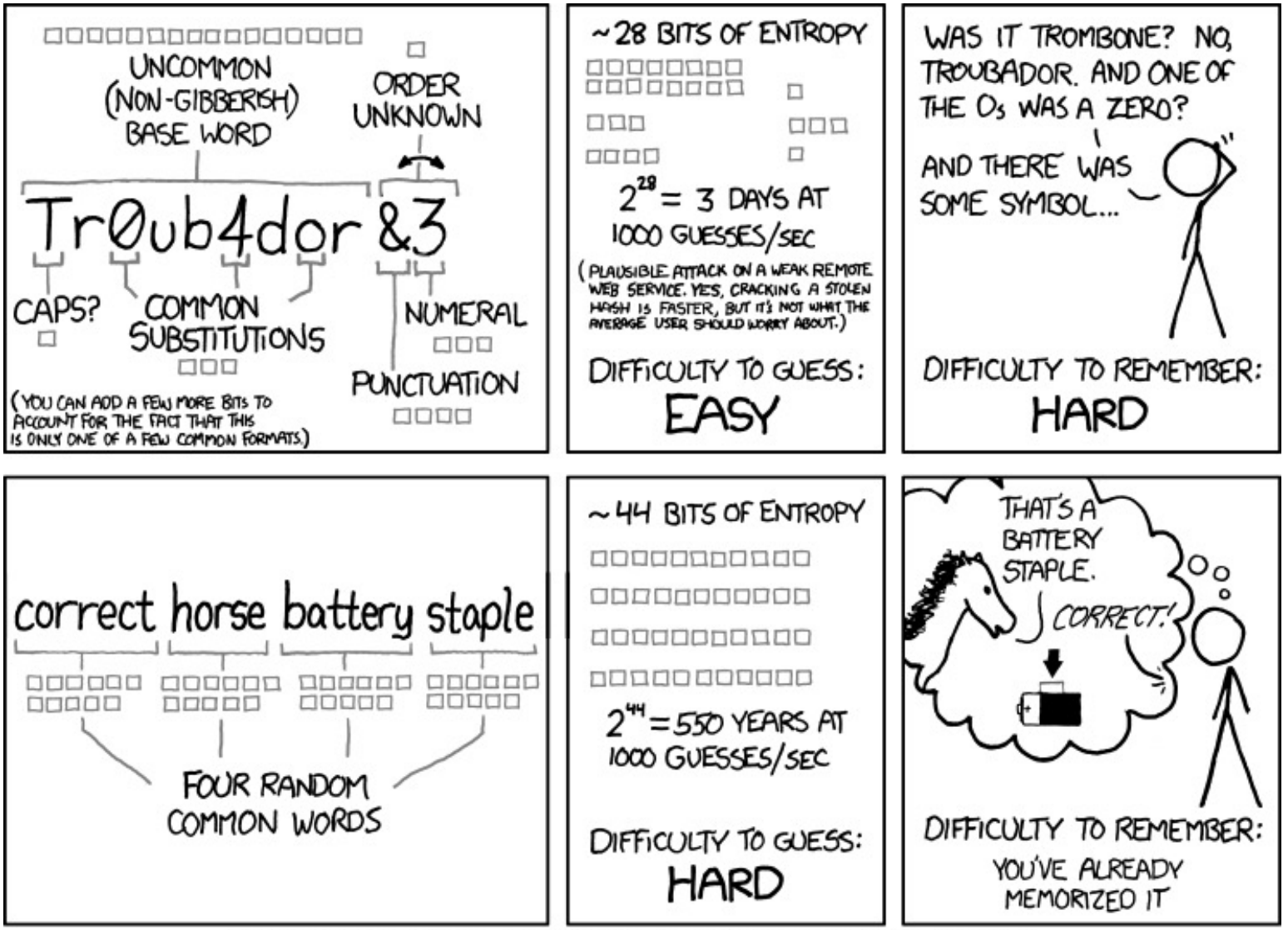
$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

  – Or alternatively:

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

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.



RockYou.com: 21.1 bits of entropy

https://xkcd.com/936/

Example: calculating entropy

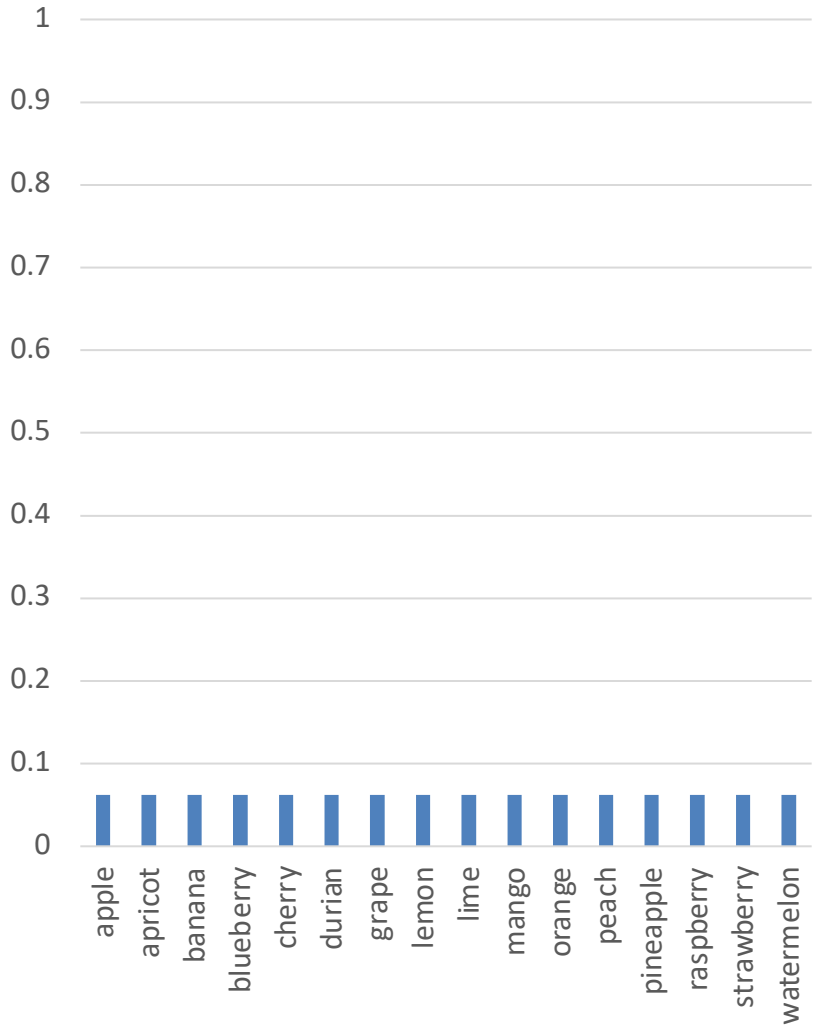
- Suppose we have a dictionary of 16 words.

- 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

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

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

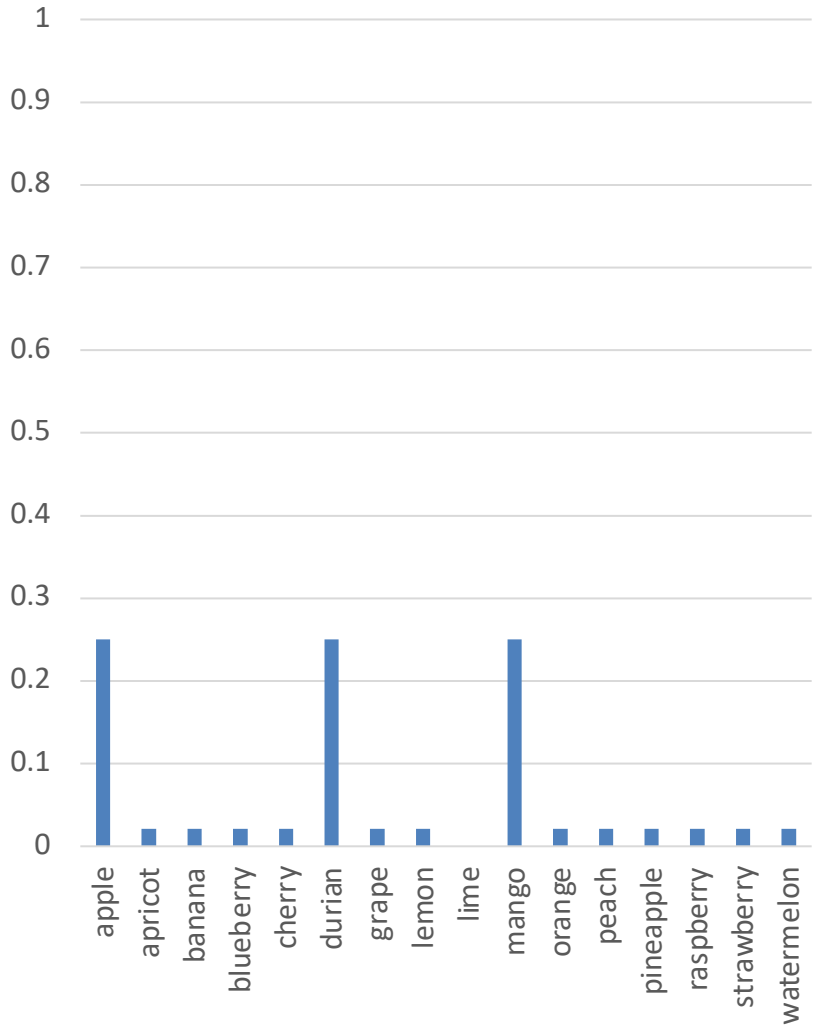
If you are trying to guess the password, you need to make about

2^{4-1} = **8 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

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

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

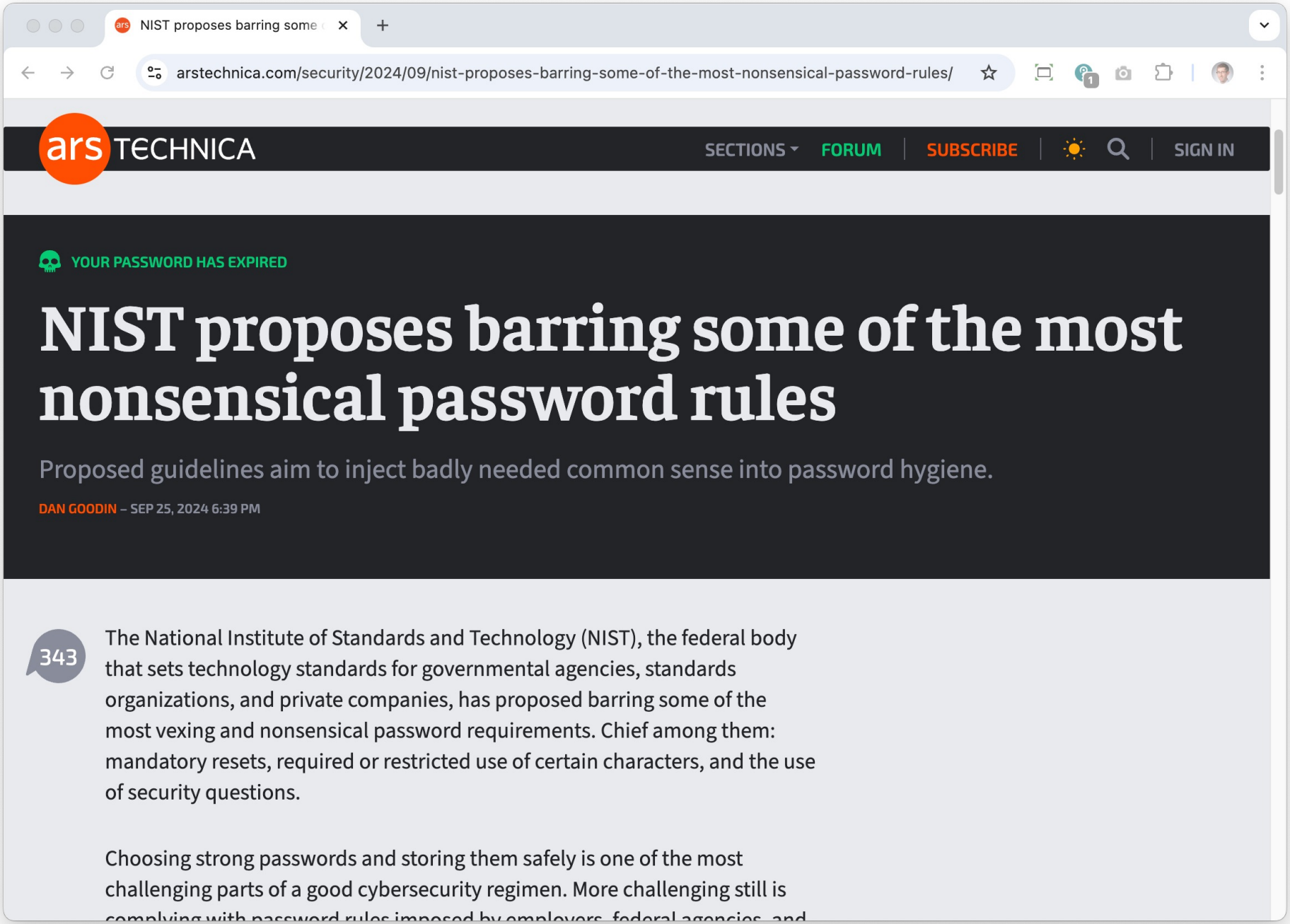
$$2^{2.8975-1} = \textbf{3 guesses}$$

on average

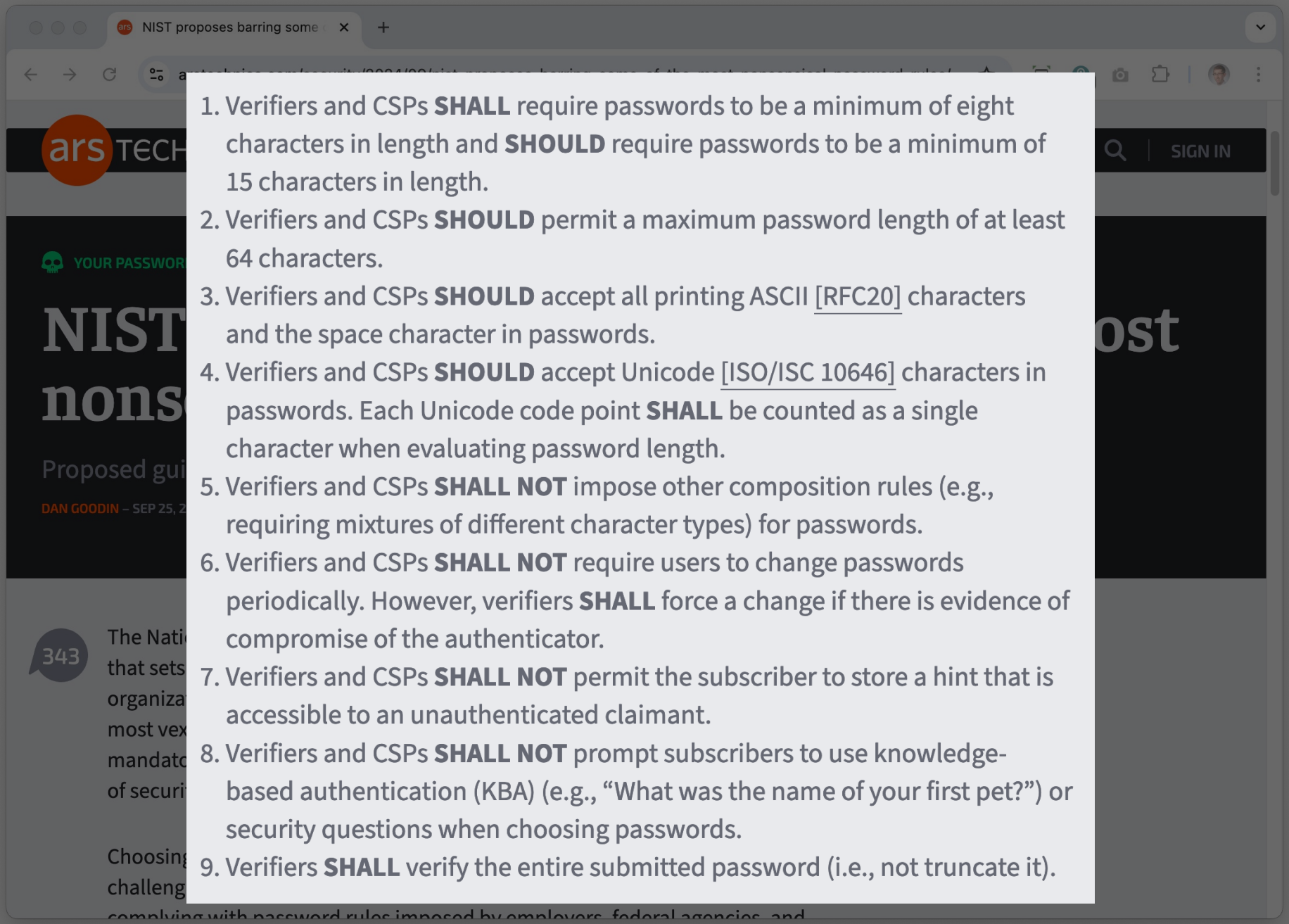
Entropy

- 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



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).

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

Password hashing

STORING PASSWORDS ON SERVERS

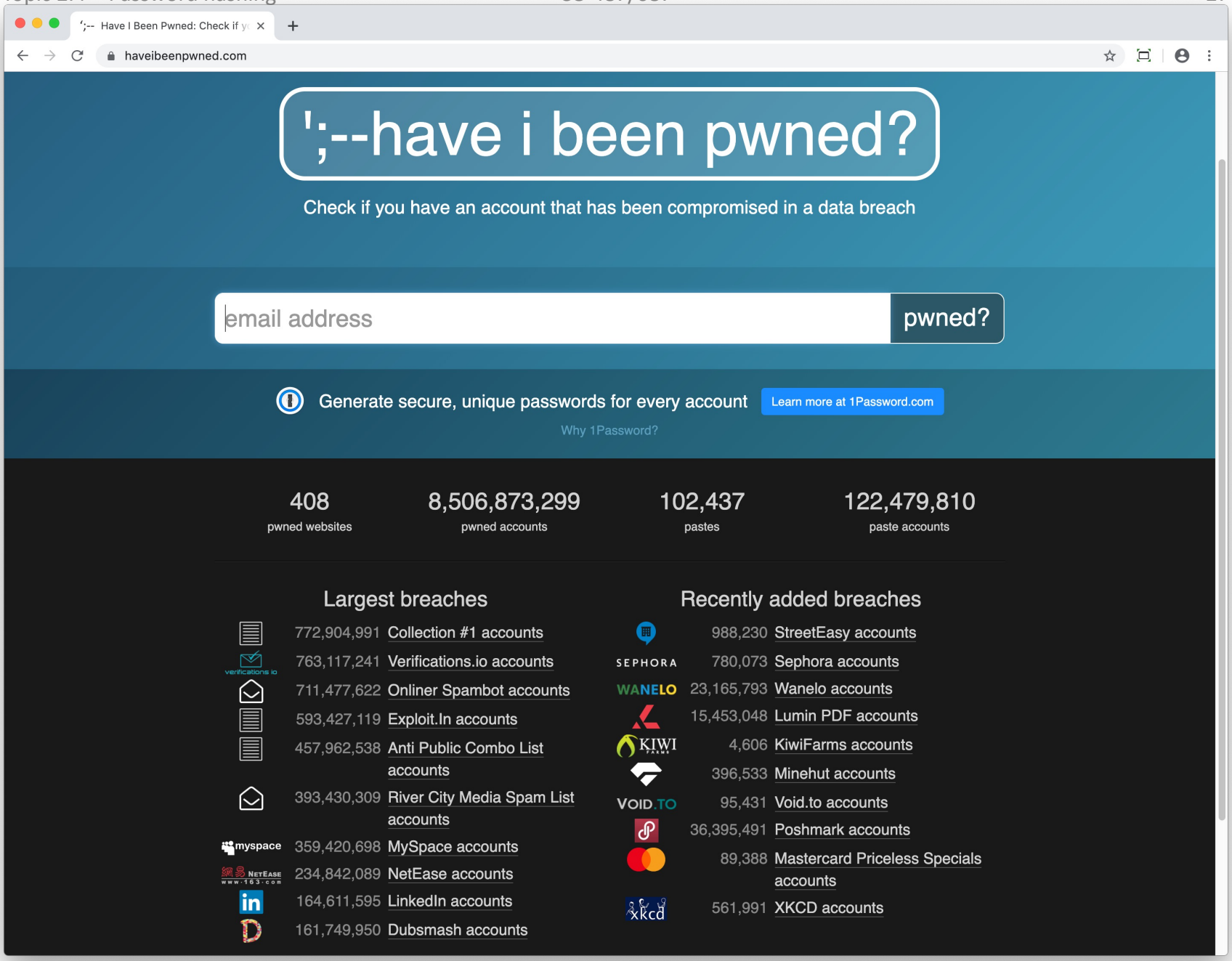
Login and registration, take 1

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



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

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

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

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

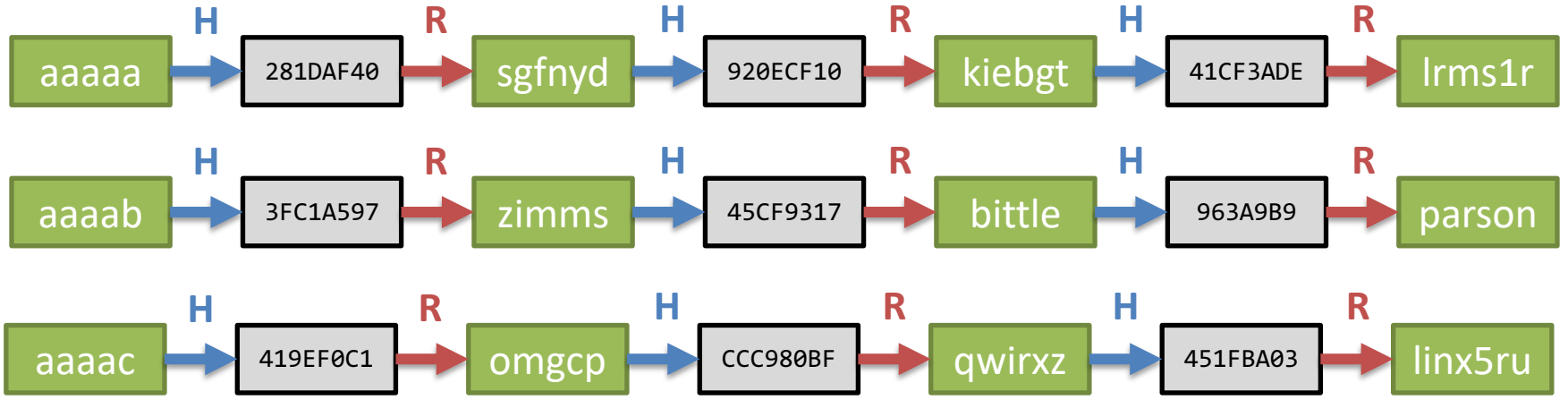
- **Rainbow tables** are an example of a time-space tradeoff using hash chains.

- 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

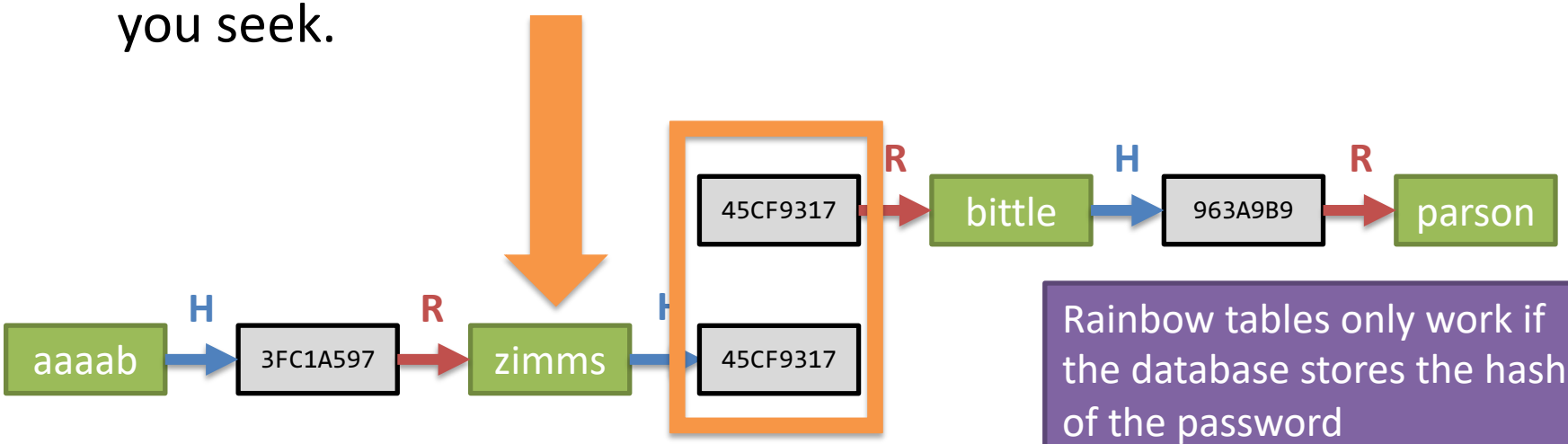


Using a rainbow table

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	lrms1r
aaaab	parson
aaaac	linx5ru



Rainbow tables only work if the database stores the hash of the password H(password).

Login and registration, take 4

Registration

1. Pick a random ≥80-bit salt
2. Store username, salt, and H(password, 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(password', salt) = 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(salt, password)
  store
  H(H(H(…H(salt, 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(password, salt) in database **where H is a password hardening function**

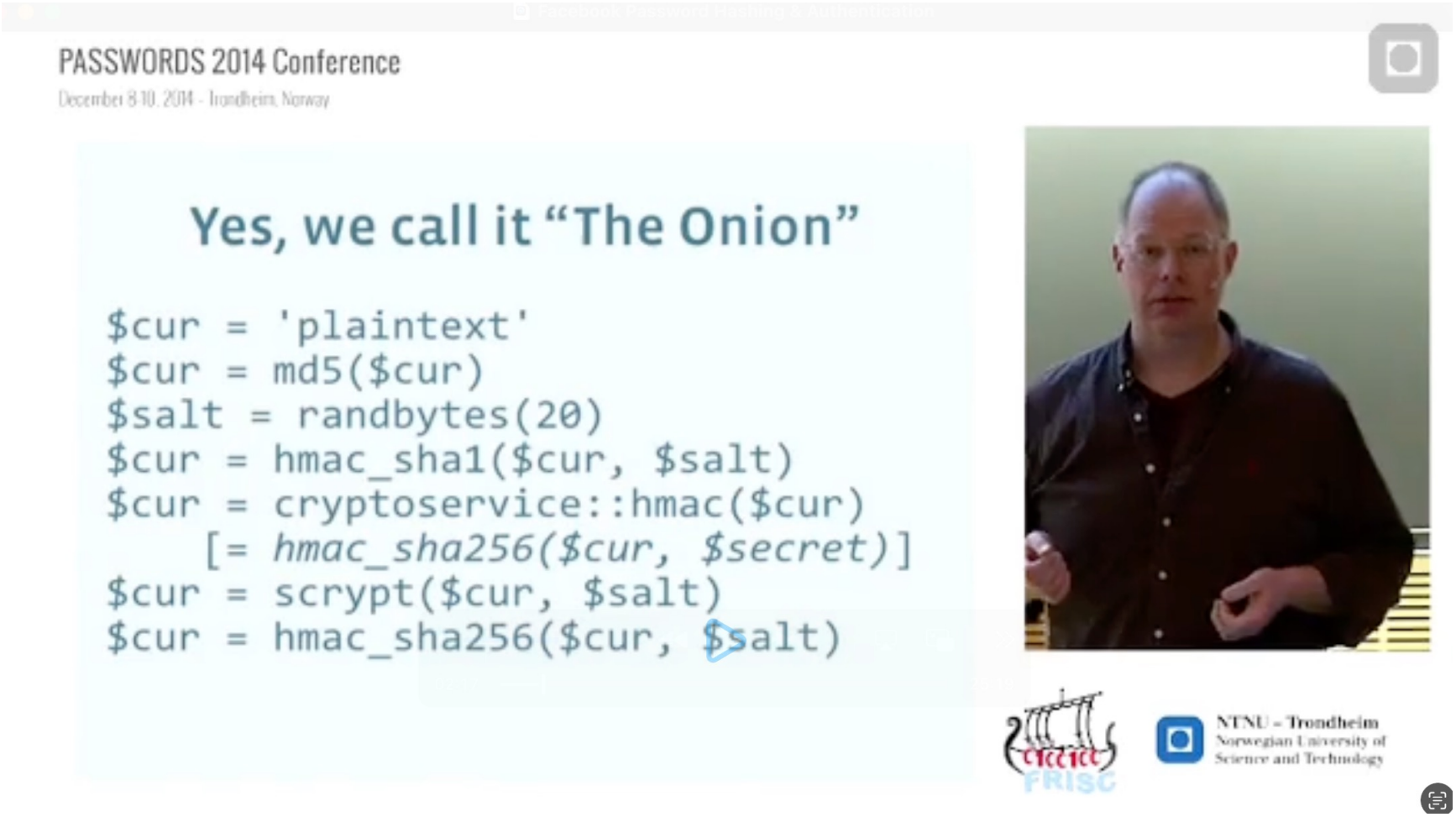
Login

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

FACEBOOK PASSWORD HASHING

Case study

Facebook password hashing as of 2014



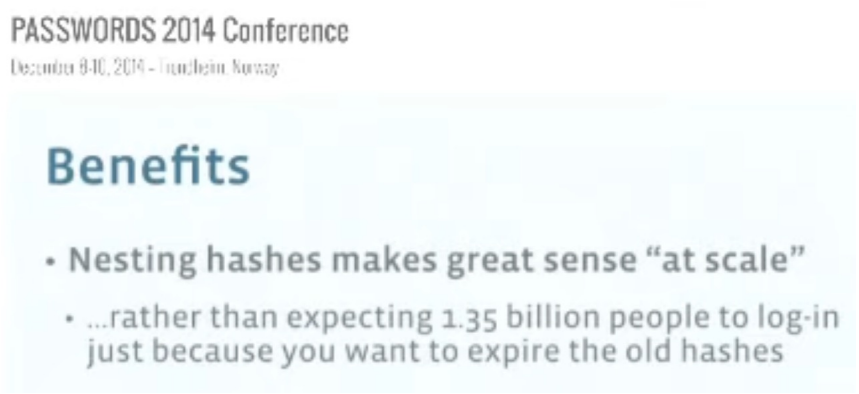
https://www.youtube.com/watch?v=7dPRFoKteIU

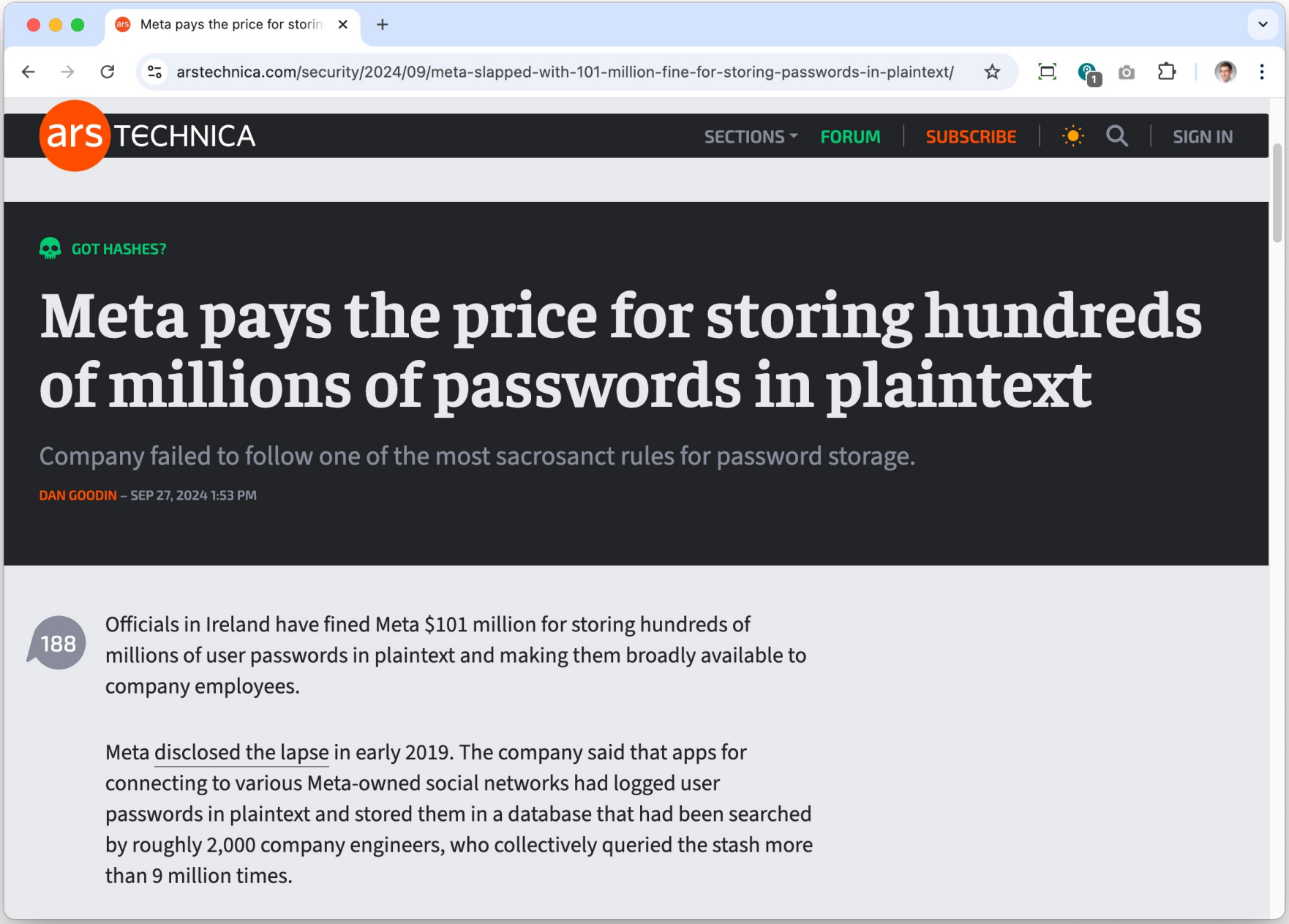
Steps of the Facebook password onion

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

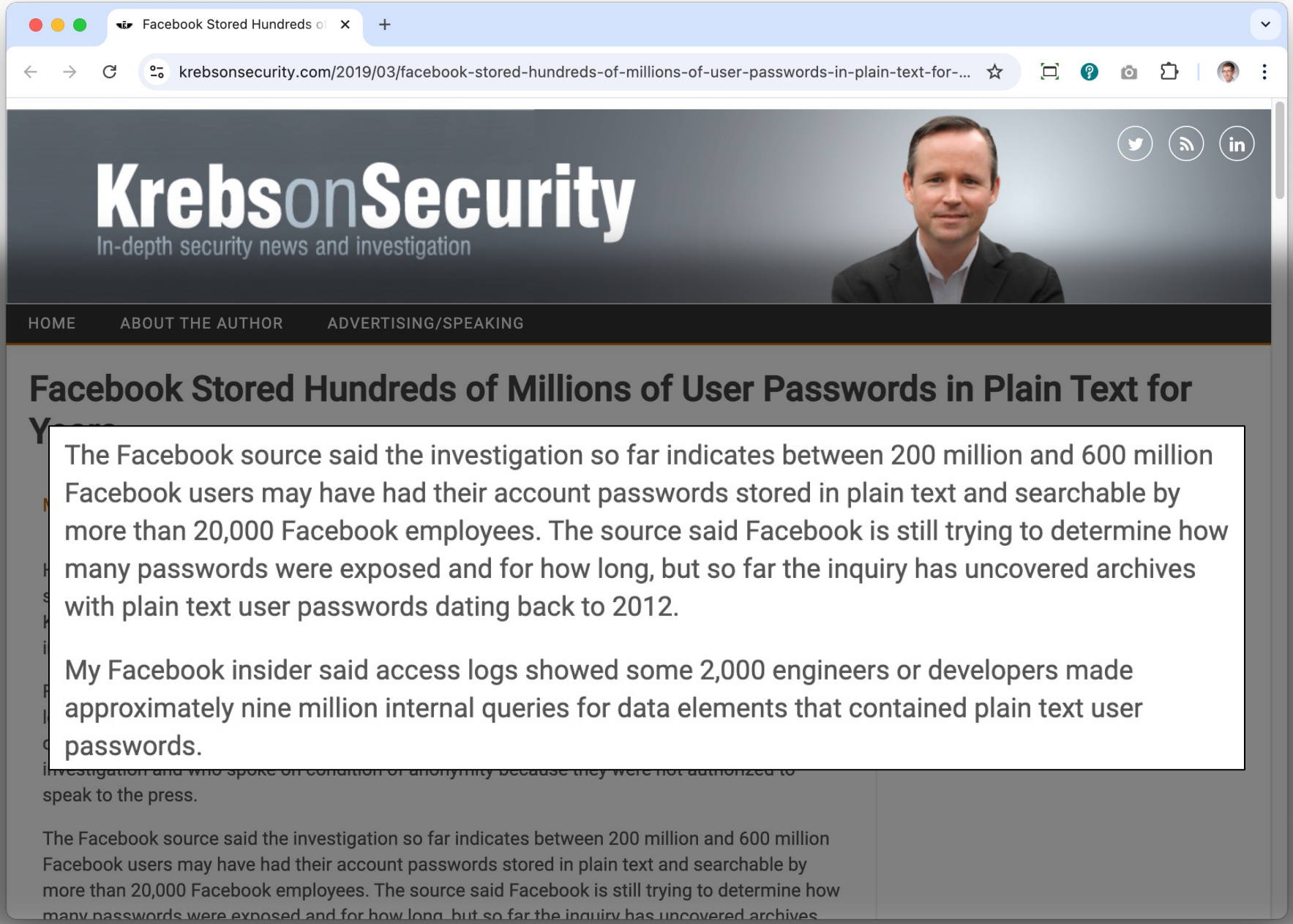
- Password hashing technology evolves over time
- Nesting "older" hashing approaches inside newer ones lets you upgrade technology without waiting for all users to log in so you can generate a new hash using the latest technology





https://arstechnica.com/security/2024/09/meta-slapped-with-101-million-fine-for-storing-passwords-in-plaintext/

https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/



https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/

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