SE 4472 / ECE 9064

Information Security

Secure Password Generation

and

Storage

Western Engineering

Secure Communication: What we can do far

- Alice and Bob can establish a shared secret
  - Confidentiality
  - Key exchange (DHE, ECDHE)
- Alice and Bob can use the shared secret to encrypt data
  - Confidentiality
  - Block ciphers, (AES, ChaCha20)
- Alice and Bob can check the data hasn't been modified
  - Integrity
  - MACs (e.g., GCM, Poly1305, HMAC, etc)

Client Authentication

- A server authenticates itself to the client with public-key infrastructure (PKI).
- A chain of certificates tie the server's public signature verification key to their identity.
- How does the client authenticate to the server?
- Does it make sense for Alice to have to buy, manage and maintain a certificate in order to log in to her Gmail account?

Some Authentication Methods

- Something you have
  - Smartcard, token, cookie, etc
- Something you know
  - Password, passphrase, etc
- Something you are
  - Biometric

Passwords

- Pros
  - Simple
  - Inexpensive
- Cons
  - Have to generate them securely
  - Have to store them securely
  - Have to remember them

Passwords and Human Memory

- Can remember quite a bit if your life depended on it and you had nothing else to do
- Not very good in practice
- 56-bits after special training
- What kind of security levels do you need?

Threat Scenarios

There are two main ways hackers can bypass/recover passwords:

- Online attacks
  - Attacker tries logging in a bunch of times
  - Mitigation: limit login attempts (simple!)
- Offline attacks
  - Attacker can make password guesses offline (no limits)
  - E.g. breached password database
  - Mitigation: password hashing (less simple)

User-Chosen Passwords

- Humans are terrible at mental entropy generation
- Trade-off between being memorable and being secure
- Common phrases (e.g., abcde, 12345)

Password Top 10

From the RockYou.com password database breach in 2009 (accounting for 2% of some 32 million passwords):

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

System Assigned Passwords

- Better entropy, harder to remember
- Middle ground: user-chosen, system “approved”
  - Password requirements (must have one digit, one special character, etc)
  - Password strength meters
  - Does it really help? If I want my password to be 12345 but you force me to use letters and special characters, maybe I’ll just choose :
    - !234S, or
    - The password I was going to choose anyway, plus “1” at the end
- People just write it down

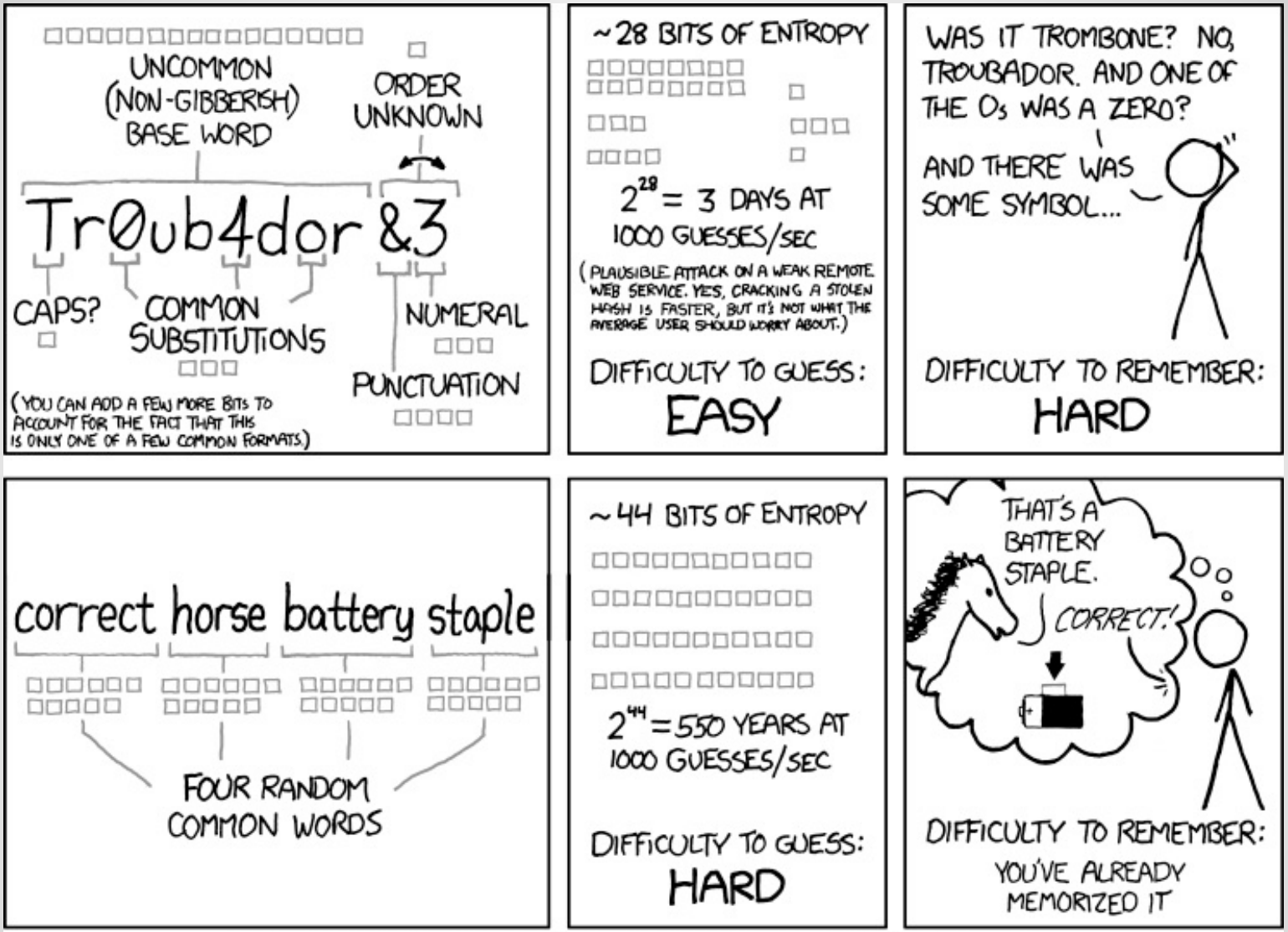
Picking Passwords: Entropy

- How can you measure the strength of a password?
- Upper bound: assume password if chosen independently and uniformly at random from a known password space
  - E.g., random 10 character lower-case alpha password: $\log_2(26^{10})$ = about 47 bits
  - E.g., random 4 words from a 1000 word dictionary: log2(1000^4)=40 bits
  - Is a random 10 character/4 word password memorable?
  - Is 47 bits cryptographically secure?
  - You can see the dilemma…

Current Guidelines

- 12 characters minimum
- No more complexity requirements (e.g. at least one upper case, lower case and special character)
- Longer and simpler passwords are better than shorter, more complex ones
- Passphrase: “I like to eat pizza every Thursday for dinner” -> iltepzevThfd
- 4-5 random words (correct battery horse staple)

https://www.canada.ca/en/government/system/digital-government/online-security-privacy/password-guidance.html



http://xkcd.com/936/

Password Databases

- Passwords have to be stored somewhere
- Have to be accessible to web-facing servers in order to perform login verification
- What happens if server gets hacked?


ONE DOES NOT SIMPLY
STORE A PASSWORD
imgflip.com

Password Hashing

The idea: don't store the password, store the hash

- **Setup**: User *u* choses pwd *p* and sends to server. Server computes *hp*=Hash(*p*) and stores (*u,hp*) in the password database
- **Verification**: Someone claiming to be user *u* submits (*u,p'*) to server. Sever looks up (*u,hp*) and checks if Hash(*p'*)=*hp.* If so, login is accepted.

Salting

Problem: If you just hash the password, then the same password will always map to the same hash. Then if Eve crack's one user's password, she will have cracked everyone else who uses the same password.

Attacker can just store all the passwords in a **dictionary** or **rainbow table**.

Idea: use a random "*salt*" *s* (similar to an initialization vector): Password hash becomes *hp*=Hash(*p*||*s*). Store (*u,hp,s*)

This way if two passwords equal, e.g., "12345," if they have different salts, they will have different hashes, and the dictionary is useless

Key Stretching

Problem: hashing is fast (think billions of hashes per second on a single GPU). Even with salted hashing, you can still tear through the most common password choices very quickly.

Idea: make hashing the password *slow*. Client and server can tolerate taking e.g., 1s to check a password, but hurts attacker if they have to make millions and billions of guess each taking 1s.

PBKDF2

Password-based Key Derivation Function: Iterative hashing with user chosen number of iterations:

$$hp = \text{PBKDF2}(\text{Hash}(), p, s, iter, klen)$$

Execution:

$hp_0 = \text{Hash}(p \,||\, s)$

$hp_1 = \text{Hash}(hp_0)$

…

$hp_{iter} = \text{Hash}(hp_{iter-1})$

Output $hp = hp_{iter}$