SE 4472 / ECE 9064

Information Security

Week 13:

Secure Password Generation

and

Storage

Fall 2015

Prof. Aleksander Essex

Western Engineering

Secure Communication: What we can do far

- Alice and Bob can establish a shared secret
  - Confidentiality
  - Key exchange (DHE, ECDHE, RSA-OAEP, etc)
- Alice and Bob can use the shared secret to encrypt data
  - Confidentiality
  - Block cipher, stream cipher (AES, 3DES, CAST, etc)
- Alice and Bob can check the data hasn't been modified
  - Integrity
  - MACs (e.g., CBC-MAC, HMAC, etc)

Authentication

- What about authentication?
- We've looked at signatures, but we need more
  - How do parties get verification keys?
  - Public-key infrastructure, certificates, etc we'll cover more in future lectures)

Authentication

- Let's talk about web authentication
- Suppose Alice is a client, Bob is a server
  - How do they authenticate to each other?
  - When does Alice need to authenticate to Bob?
  - When does Bob need to authenticate to Alice?
- Idea: Bob can use PKI to authenticate to Alice
- Does it make sense for Alice to have to buy, manage and maintain a certificate to log in to stuff on the web?

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

- Online attack
  - Attacker tries logging in a bunch of times
  - Mitigation: limit attempts
- Offline attack
  - Attacker has some password-protected data and can conduct attempts offline
  - Mitigation:

User-Chosen Passwords

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

Password Top 10

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

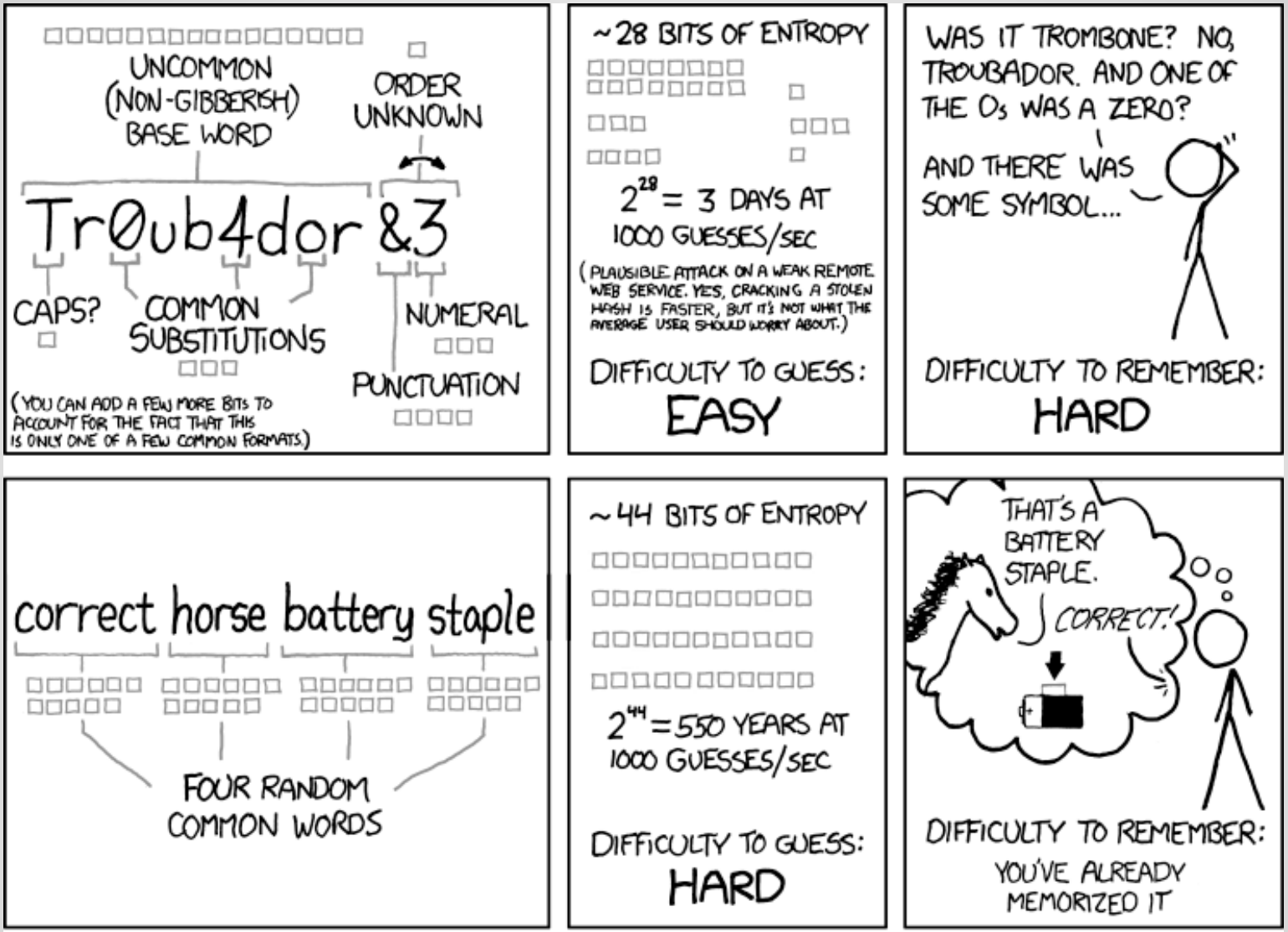
1. 123456
2. 12345
3. 123456789
4. password
5. iloveyou
6. princess
7. 1234567
8. rockyou
9. 12345678
10. abc123

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
  - Is a random 10 character password memorable?
  - Is 47 bits cryptographically secure?
  - You can see the dilemma…



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 rooted?
  - All your password are belong to us
  - Example: Bell (January 2014)


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.

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

This way if two passwords equal, e.g., "12345," if they have different salts, they will have different hashes.

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 $hp = hp_{iter}$

Memory Hard Functions

Problem: PBKDF2 and related key stretching algorithms (like bcrypt) are highly parallelizable. Attacker could just buy lots of GPUs. Sound crazy? See Bitcoin.

Idea: make password verification take lots of time and lots of memory. CPUs have lots of memory. ASICS, GPUs, FPGAs, not as much. Make it more expensive for attackers.

scrypt

- Pros:
  - Memory-hard
  - Becoming popular
- Cons:
  - New (and not as well understood)
  - Complicated
    - Harder to analyze
    - Harder to implement

What should *you* use today?

- PBKDF2
- bcrypt
- scrypt

What should we be working towards?

- New memory hard functions
  - Things that are cheaper to do on a CPU that a GPU/ASIC/FPGA

- Why not use passwords in conjunction with a key?
  - Risk if server gets rooted
  - Hardware security module (e.g., a hardware-secured encryption oracle)
  - Send password to HSM, it returns ciphertext. Compare to stored value. Pro: attack at minimum requires physical access. Con: admin needs to buy hardware, configure setp