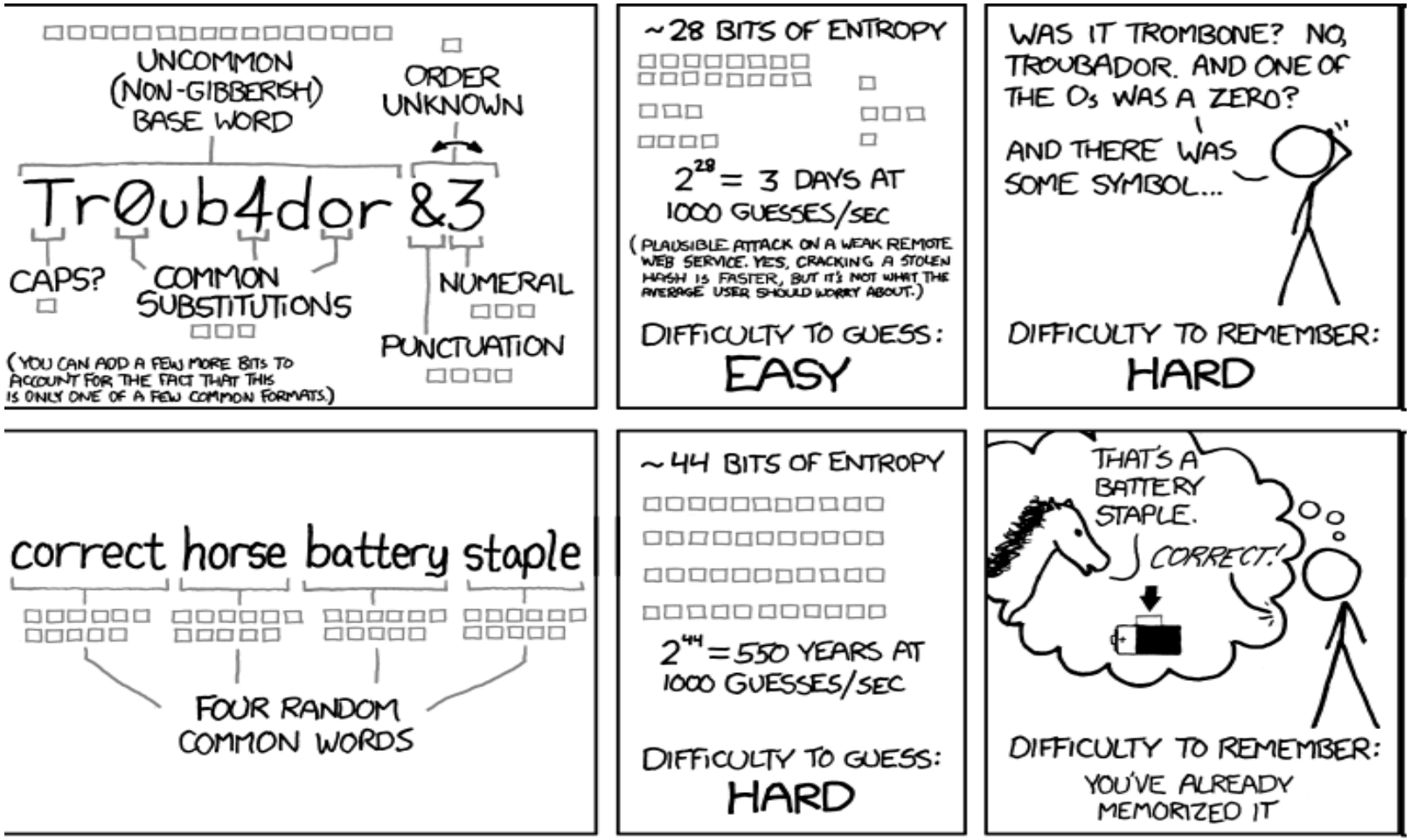
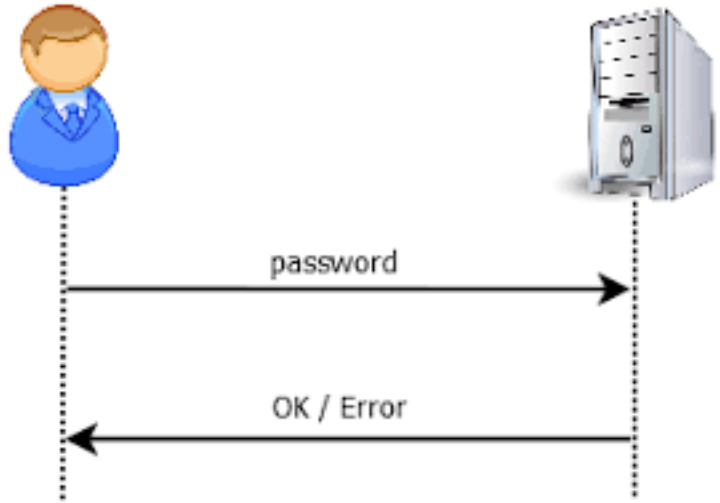
Password-Based Authentication



Password-Based Authentication

- Passwords are susceptible to *keylogging*, brute-force and dictionary attacks and can too often be obtained or guessed using social engineering techniques.

- The success of a brute-force attack depends on the size of the password/pin.
  - A 4-digit pin can be guessed after 5000 tries ((9999 – 0000)/2).
  - A 6-digit pin takes 500,000 guesses.
  - A 6 character password using lower-case letters takes 150 million guesses.



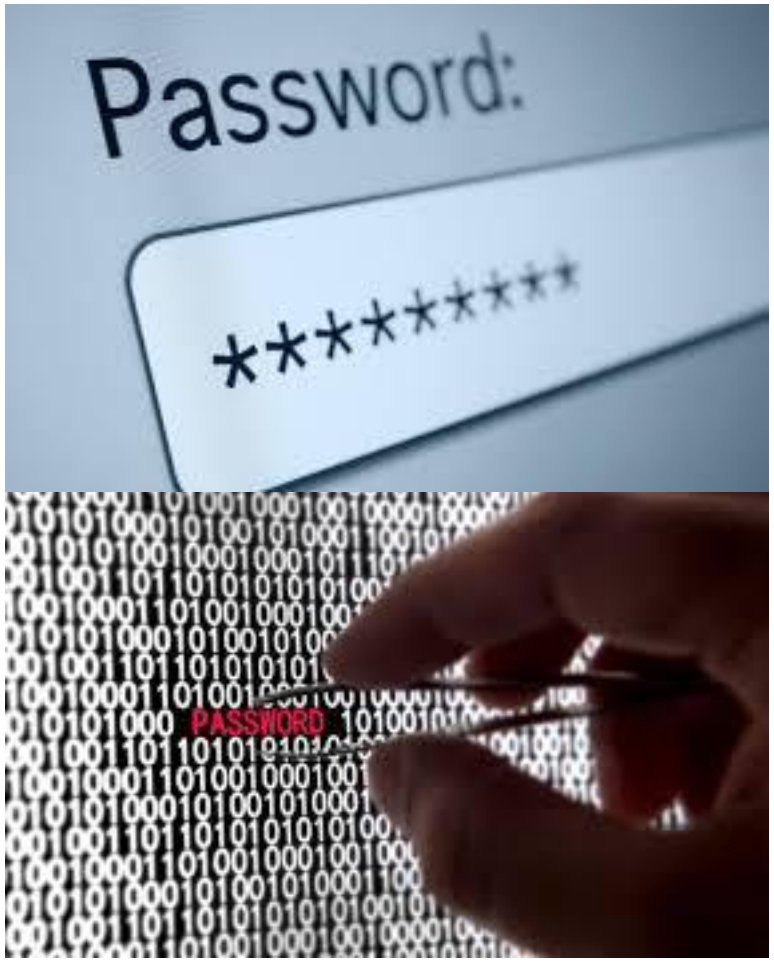
Password-Based Authentication

- These passwords can be 'cracked' easily using existing software that automates the login process.

- Dictionary attack - SSH-brute.c
  - A 6-character password using upper and lower case letters, numbers and symbols (62 possible characters) will take about 28 billion guesses.
  - Increasing the number of characters to 8 improves things by a factor of 256,000 to 110,000 billion attempts needed.
  - Passwords of this size can still be guessed, but it takes an inconvenient time.

Password-Based Authentication

- If the password can be guessed, the speed of cracking increases dramatically.
- A dictionary attack may take as few as 85,000 guesses (171,000 words in the Oxford English Dictionary).
  - Try HashCat (in Kali VM)
  - Bad guys know how people "disguise" their passwords or append them to satisfy password policies. Most common modifications are scriptable.
- Most passwords are now 8 characters
  - Default passwords are by design easy to remember (short)!



Password-Based Authentication

Don't write the password down.

http://www.theregister.co.uk/2005/08/10/kutztown_13/

If you really have to write it down, keep it in your wallet with your money. Don't write down what it's for.

Password storage?

SIMS synchronizes passwords for all accounts

- password re-use

Browsers store passwords

- in plain text

Password Managers

- only as safe as the master password.

Allow for *escrow*

Password Salt

- One way to make the dictionary attack more difficult to launch is to use salt.
- Associate a random number with each userid.
- Rather than comparing the hash of an entered password with a stored hash of a password, the system compares the hash of an entered password and the salt for the associated userid with a stored hash of the password and salt.

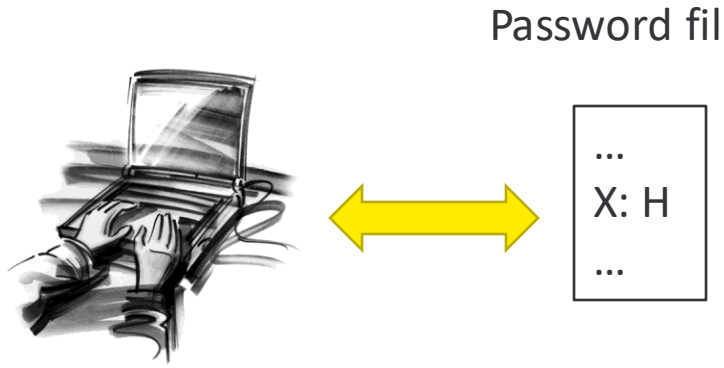
How Password Salt Works

- **Without salt:**

➢ 1. User types userid, X, and password, P.

➢ 2. System looks up H, the stored hash of X's password.

➢ 3. System tests whether h(P) = H.

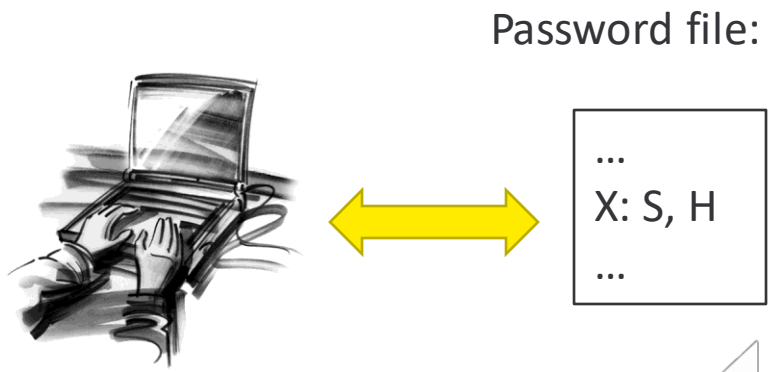


- **With salt:**

1. User types userid, X, and password, P.

2. **System looks up S and H**, where S is the random salt for userid X and H is stored hash of S and X's password.

3. System tests whether h(S||P) = H.



Stealing Passwords

- Passwords are obtainable by social engineering methods or by sniffing unencrypted network traffic (Telnet, HTTP, FTP).
  - People choose a password they can remember, and then **re-use** it for several accounts.
  - May be the name of a child, pet or relative.
  - May be used for plain text traffic (to http:// web site form), sniffed, and then tried on secure (https://) accounts.
- Even a password used with a secure web site may be obtained if it is stored in plain text in a vulnerable database or as an *unsalted hash* (look up *rainbow tables*).

Password policy

- To minimise the risk of a password being guessed, each password should be
  - long,
  - random,
  - different.
- A password policy can be used to develop an algorithm for re-creating different passwords for each account:

```
e.g. unique password = f(master password, site name)
f("password", "gmail.com")
{
 first 3 letters of password + every second letter of site + "x"
} //pasgalcmx
```

- Mixtures of case and character substitution can be used as well.
  - e.g. `mYPa55w0rd`
- An entirely random password can be stored in a text file and copied into web forms as needed.