Passwords – the server side

A tour of decreasingly bad ideas
regarding server-side password handling.

Thomas Waldmann @ EuroPython 2013

Disclaimer

I am *not* a crypto or security expert,
just a caring developer taking a look
from a practical point of view.

About me

- Thomas Waldmann
- doing Python stuff since 2001
- long-term MoinMoin Wiki core developer / project admin
- loves FOSS, Linux, Python and Wikis
- own small company near Stuttgart, Germany
- admin stuff, services, training, SW development

Why this talk?

- MoinMoin 1.9.5 issue CVE-2012-6081
- remote code execution
- some wiki sites compromised, python wiki erased
- incident management of that / aftermath
- awareness raising
- responsibility for users' (passwords) safety
- shit *will* happen – be prepared, have a plan!

How to authenticate users/clients?

- **userid + password → covered in this talk**
- Other ways → not covered here
  - SSL client certificate
  - LDAP
  - OpenID
  - OAuth
  - Mozilla Persona
  - CAS
  - …

userid + password

- You will need to store **user profiles** somehow (in a database, in some on-disk file, ...).
- The profile usually contains:
  - username
  - something (P) you need to verify the clear-text password (CTP) the user enters at login time
  - E-Mail address
  - other properties / settings

Storing plain-text passwords

- P = CTP
- When registering, the user enters the clear-text password (CTP) plus other stuff (name, email).
- You just store everything into the user profile, including P.
- Popular method in the last millennium!
- But....


ONE FACEPALM IS NOT ENOUGH WHEN THE SEVENTIES ARE CALLING
memegenerator.net

CTP storage = users globally owned!

- profile storage access == knowing all userids, E-Mail addresses and passwords!
  - admins (legitimately? accidentally? evil admins?)
  - everyone else getting access (some security issue will happen, sooner or later)
- Users re-use passwords for multiple sites.
- If the userid for login is the E-Mail address, that's usually the same everywhere also.

Let's obfuscate.

- P = obfuscate(CTP)
- obfuscate == rot13, xor, base64, …
- Not quickly readable any more, but:
  - This is obscurity, not security.
  - Can easily be reversed and you have the CTP.
  - Useless.
- → Double-Facepalm applies here, too.

Let's encrypt!

- Obfuscation bad, so let's use *real* encryption:
- P = encrypt(key, CTP)
- encrypt == aes, des, … (any symetric crypto)
- To verify a password at login time, decrypt P and compare with CTP.
- But...

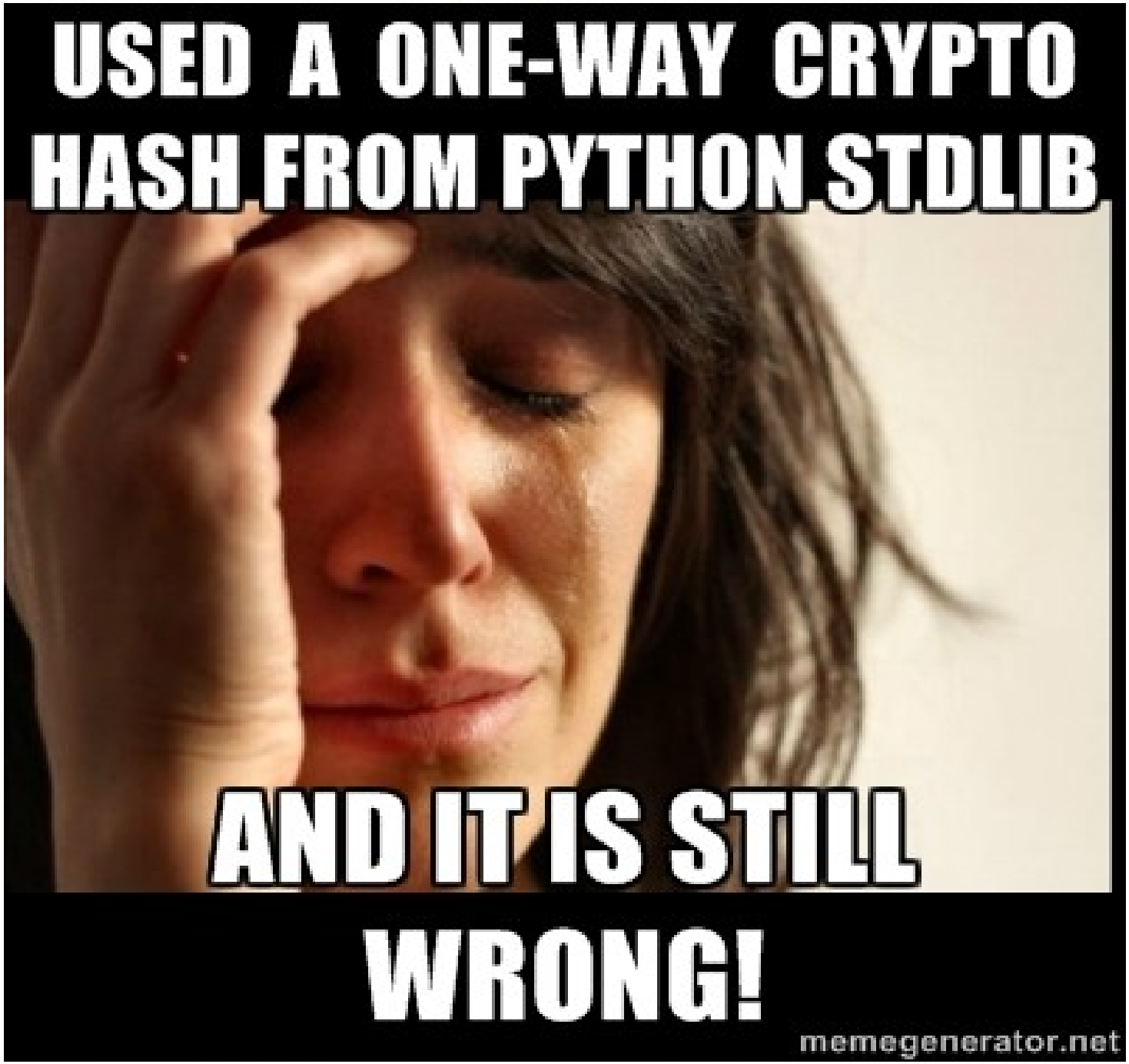

ONE DOES NOT
SIMPLY
ENCRYPT PASSWORDS
memegenerator.net

It's symetric!

- You need to have the crypto key somewhere on the server (on disk, in database, in RAM, ...) to:
  - encrypt the CTP when a new password is set
  - decrypt P to verify the given CTP at login time
- Thus: an attacker (who hacks into your server) can have the crypto key, too.
- Your encrypted passwords are as safe as your crypto key storage and your crypto algorithm → unsafe in many cases.

One-way crypto hashes FTW!

- P = cryptohash(CTP)
- cryptohash == md5, apr1, sha1, ntlm, …
- One-way crypto hashes can only be easily computed in one direction (CTP → P),
- but not in reverse direction (P → CTP).
- Popular methods, even in 2013!
- But...

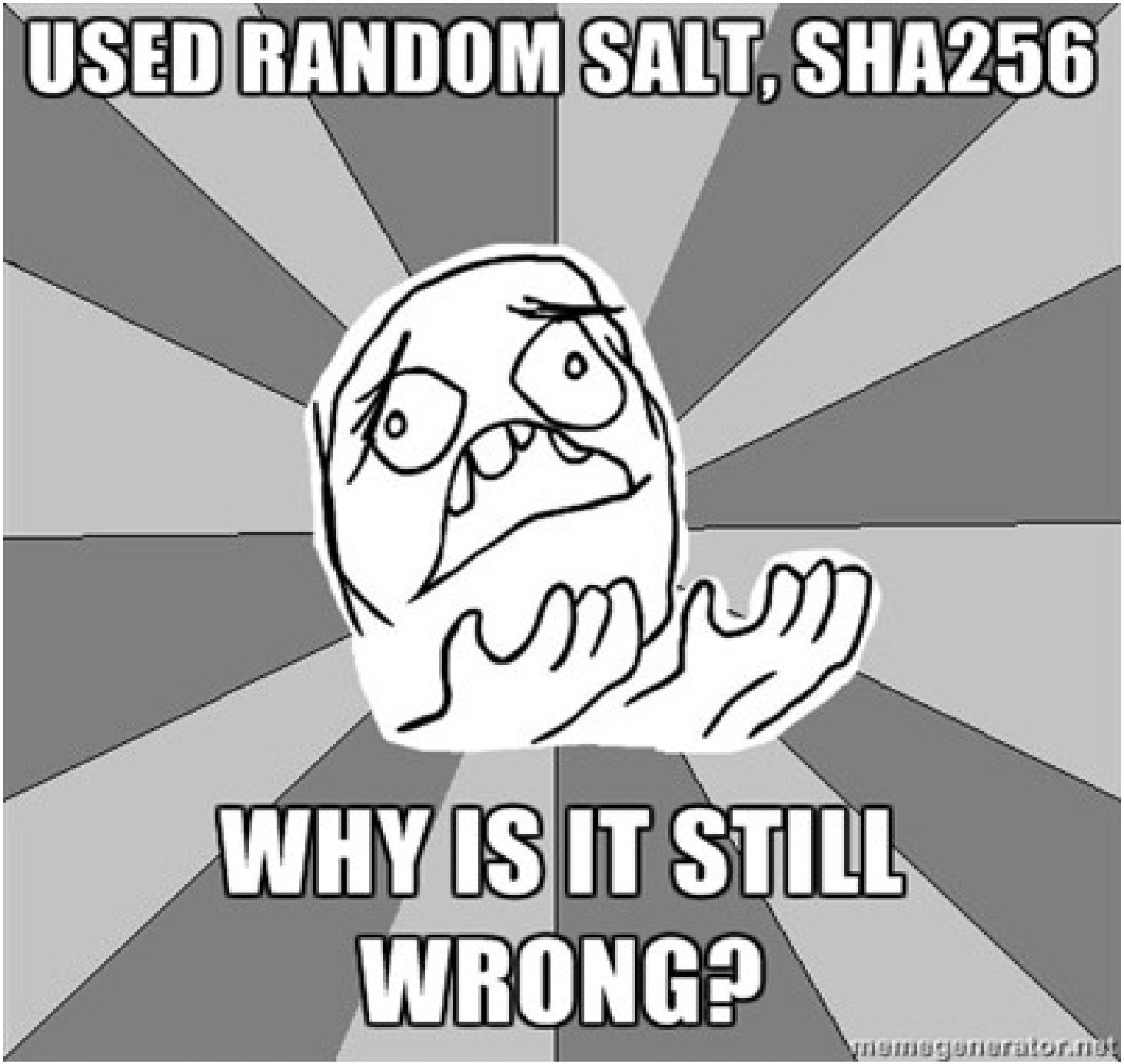

USED A ONE-WAY CRYPTO HASH FROM PYTHON STDLIB
AND IT IS STILL WRONG!
memegenerator.net

Algorithms, bitlengths, rainbows!

- same CTP → same P – bad!
- Some algorithms are weak (e.g. md5).
- Some create hashes of short bitlength.
- For some, so-called rainbow tables can get precomputed to break pw hashes just by a table lookup (P → CTP). Byebye one-way.
- So let's spoil the rainbow table based attacks, take many bits, a good algorithm and...

Add some Salt (plus Pepper?)

- salt = compute_random_bytestring(32)
- P = salt + crypto_hash(salt, CTP)
- crypto_hash = sha256, sha512, …
- same CTP, different salt → different P – good!
- (pepper is similar to salt, just kept elsewhere)
- many bits, many different salts → rainbow tables impractical (impossible?)
- But...


USED RANDOM SALT, SHA256
WHY IS IT STILL WRONG?

Who needs rainbow tables nowadays?

Computes 1.000.000.000 sha256 hashes every SECOND!

So just brute-force, start with password = “0000000”, iterate to “zzzzzzz” until your computed hash value matches the given one.

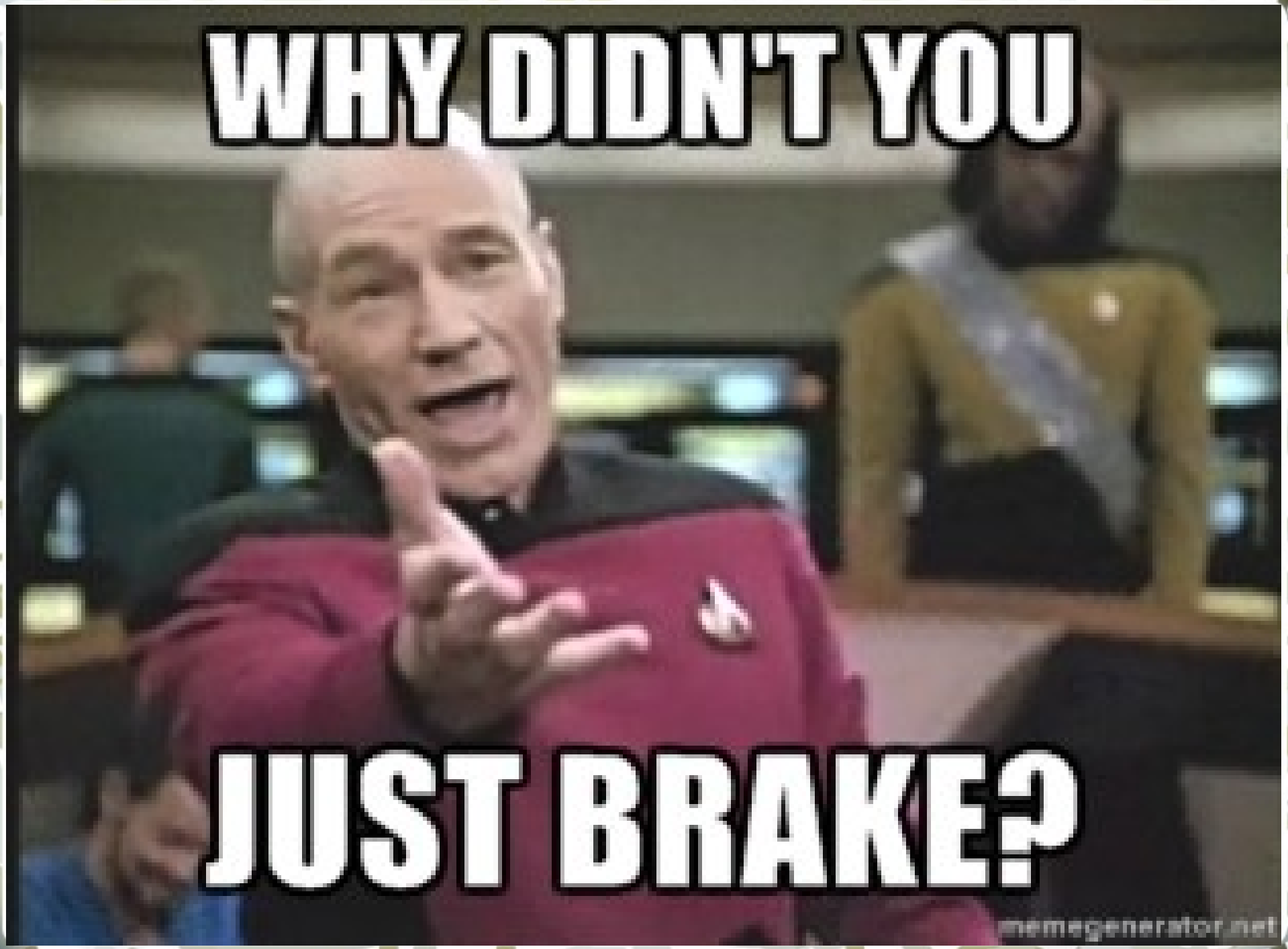
1-2 cards = owned by many gamers

1.000 cards = owned by whoever has enough money to buy

Details: http://hashcat.net/ (NVidia works too!)

Beyond simple brute force.

- Use dictionaries of words / names / passwords,
- combine them,
- mutate them using popular rules:
  - leetspeak → l3375p34K
  - insert / append /delete chars
  - add some numbers (1900..2013 maybe?)
  - do same stuff humans usually try
- Feed back broken passwords into the dictionary.
- ...


WHY DIDN'T YOU
JUST BRAKE?
memegenerator.net

NO need for speed.

- Hashes like sha* are
  - optimized for speed,
  - not designed for password storage.
- Speed is:
  - good for brute forcing,
  - not needed at login time (users tolerate 250ms).
- Thus, we need something slower, optimized for password storage needs. (we will come back to that later)

Unexpected other troubles

- For simplicity, we assume:
  - stored cleartext passwords
  - they won't ever get disclosed / stolen
- successful_login = (entered_pw == stored_pw)
- Looks OK?
- Too trivial code to have an issue?
- Nope...

Time is not on your side (1)

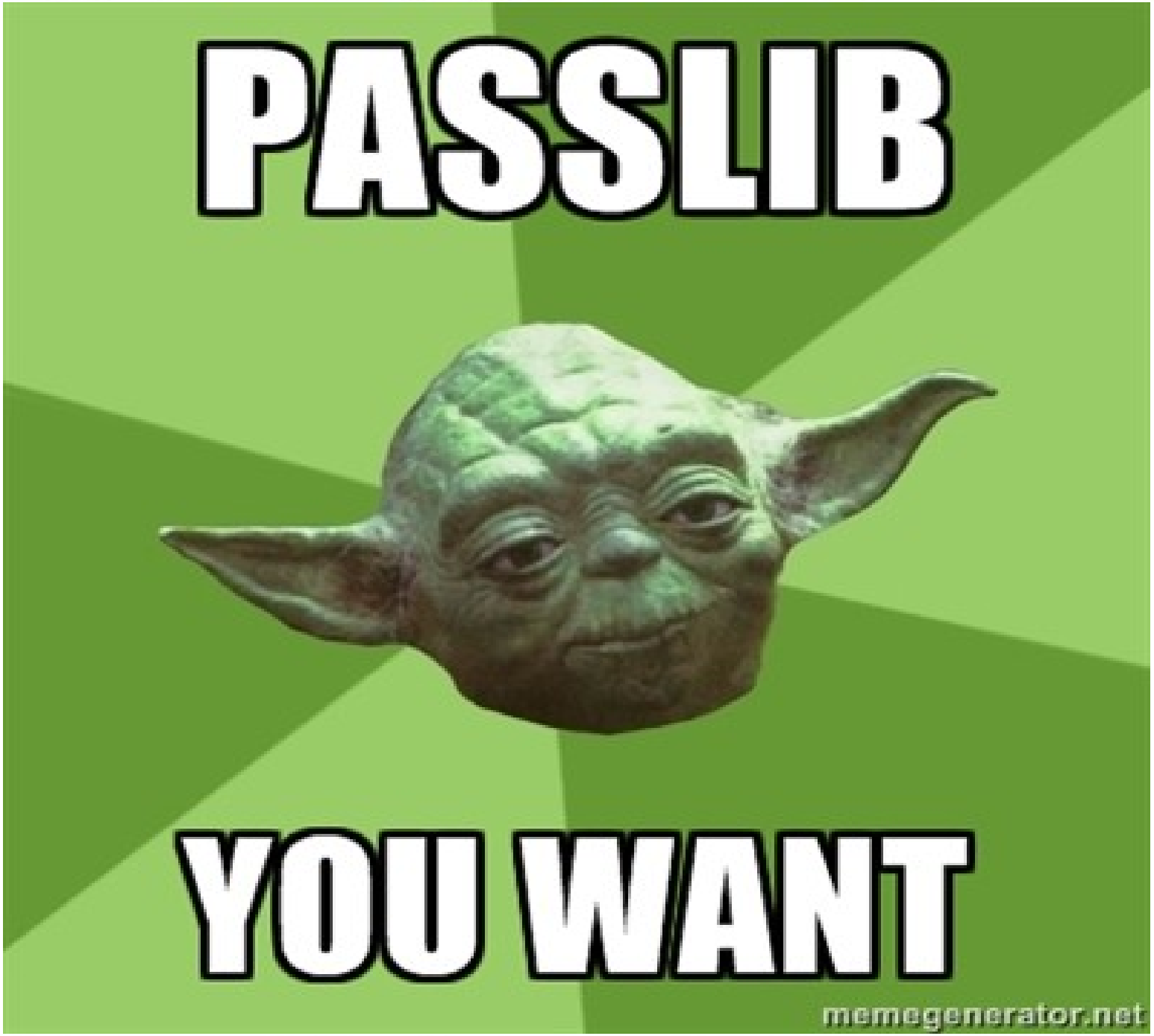
- Assume comparing one char takes time T on your CPU (left: attacker's guess, right: real pw):
  - “0000” == “1234” # False after T
  - “1000” == “1234” # False after 2T (jumped!)
  - “1100” == “1234” # False after 2T
  - “1200” == “1234” # False after 3T (jumped!)
- Python's speed-optimized “==” for strings can't keep a secret, jumps up happily for every correctly guessed character!
- → use a “constant time comparison” function.

Time is not on your side (2)

- For failed logins, you are of course *not* telling:
  - “your username was wrong”, or
  - “your password was wrong”.
- But...
  - if username not in valid_usernames: fail # after T1
  - if not validate(password): fail # after T2
- Your normal code execution timing will tell it!
- Evaluate both expressions, try to use similar code or even always adjust timing to worst-case-timing.

Lessons learned

- Don't use home-grown password code:
  - you can easily do it wrong,
  - it is work to write and ...
  - to continually review, improve and maintain.
- Use such code from a good password library:
  - done right,
  - less work, enables collaboration,
  - contribute to it
    (use it, test it, fix it, review it, improve it, …).


PASSLIB
YOU WANT
memegenerator.net

… or other good lib / framework

- Django
  - has own code for password handling
  - does not use passlib (why not?)
  - uses strong algorithm (pbkdf2) by default
- Specific hash code (flexibility?)
  - py-bcrypt (not pure python)
  - python-pbkdf2 (maintenance?)
- Any other good stuff suggested by audience?

passlib

- http://code.google.com/p/passlib/
- Main author: Eli Collins, BSD License
- Pure Python (2 and 3) for all password needs.
- Offers multiple strong algorithms via one API:
  - bcrypt (needs additional C code / binaries)
  - pbkdf2
  - sha512_crypt (not the same as sha512!)
- Written with care, easy to use, good docs, ...

passlib “hello world”

```
# fastest route is to use a preconfigured context:
from passlib.apps import custom_app_context as pwd_ctx

# creating a password hash:
hash = pwd_ctx.encrypt("somepass")  # <- normal user
hash = pwd_ctx.encrypt("somepass", category="admin")

# verifying a password:
ok = pwd_ctx.verify("somepass", hash)

# hash looks like:
# $5$rounds=160397$8kDQgoNHYhPc5lGw$jLsSM/87lvESBACxg0e7VuTTw4m5s3k0R6OjZYuB0/2

# identifying a hash:
algo = pwd_ctx.identify(hash)

# algo == 'sha256_crypt'
```

passlib offerings

- The strong algorithms offered by passlib are:
  - slow (~1.000.000 times slower as sha)
  - tunable (e.g. rounds parameter)
  - configurable, with good defaults
- constant time comparison done automatically
- hash upgrades done comfortably
- password and random byte string generator
- also weaker popular algorithms for compatibility

passlib - more control and options

```
from passlib.context import CryptContext

# do this once, import pwd_ctx whereever it's needed:

pwd_ctx = CryptContext(
    schemes = ["sha512_crypt", "hex_sha256", ],

    default = "sha512_crypt",
    deprecated = [“hex_sha256”, ],

    # vary rounds randomly when creating new hashes
    all__vary_rounds = 0.1,

    # set the number of rounds that should be used
    sha512_crypt__default_rounds = 100000,
)
```

passlib – hash upgrade

```
password = “foo”

# deprecated hex_sha256 hash:
old_hash = '2c26b46b68ffc68ff...fa0f98a5e886266e7ae'

ok, new_hash = pwd_ctx.verify_and_update("wrong", old_hash)
# → (False, None)

ok, new_hash = pwd_ctx.verify_and_update(password, old_hash)
# → (True, '$6$rounds=92367$GC3oiYRJ0oqZQI3i$kDHkk...jdAVi00')
# new_hash now has the upgraded hash (sha512_crypt)

if new_hash is not None:
    save_to_profile(new_hash)

ok, new_hash = pwd_ctx.verify_and_update(password, new_hash)
# → (True, None)
```

Weak hashes for compatibility?

- You could import these weak old md5 (or other) hashes from some legacy system to have happy users who don't need to reset their passwords.
- You could upgrade the hashes at login time to something really strong.
- But: if some users never log in, you keep the weak hashes forever, endangering these users.
- → Either just DON'T or use double-hashing...

Double Hashing, the rough idea

Make sure that the legacy system did not have bigger security issues or even security breaches, otherwise taking the old weak hashes is no option anyway as you have to consider them already being exposed/broken.

If you made sure:

Import weak hashes WH, but hash them again using strong algorithm: P = strong(WH)

Login time: you have the CTP available (*must be impossible to give WH*), so re-write: P = strong(CTP)

Run production systems in this mode for a while → frequent users will have a strong P then.

To get rid of the double-hashing and the inner weak hash:

invalidate all remaining double-hashes (will only affect infrequent users)

send these users a password-reset link, publish contact info for cases of trouble

user still interested → will just reset the password

not interested any more → won't do anything

in any case → no double-hashes in password storage any more

Disable support for double-hashing in the site configuration.

Later: first deprecate, then remove support for double-hashing in the software to simplify it.

Acceptable Passwords?

- Too simple / short / common passwords will get cracked instantly, no matter how good your password hash algorithm is. Thus:
  - give instant feedback about password strength
  - allow very long passwords
  - accept all characters, so users can generate passwords without running into issues
- If you offer a password generator:
  - avoid chars **1lIO05S6G8B**... in random password for better readability

PW Strength Measurement (1)

- Simple checks are not good enough:
  - length check: aaaaaaaaaa
  - set size check: 12345678
  - usual upper/lower/digit/punctuation requirement: Thomas.Waldmann.2013
  - …
- But what else can we use?
  - dictionaries are language dependent and big
  - check shouldn't take too much time / resources

PW Strength Measurement (2)

- Self-entropy / information content (Shannon)?
  - not good for PWs as they are relatively short, so the whole original character set isn't as obvious as in a much longer text.
  - all symbols are equally difficult - unrelated to brute-forcing character ranges (like a..z, A..Z, 0..9, special, non-ascii chars)
- Idea: modified shannon self-entropy, consider required brute-force scan-ranges.
- Your idea?

PW Strength Measurement (3)

- Likely you can never tell if something is a good password, so rather say “maybe good”.

  E.g. “correct horse battery staple” (famous due to xkcd) is not a good password any more.

- Avoid the worst passwords by a strength meter.
- Educate users.

(Interactive) Login Behaviour

- Slowing down / disallowing login after bad attempts:
  - useless against distributed attacks
  - in some scenarios: risk of DOS
  - one would notice if someone tries to get in
  - except if attacker is trying slowly/distributed
  - won't help if someone gets the hashes
- If you use some slow & strong hash algorithm:
  - tune the rounds, so it is neither too slow nor too fast
  - frequent “logins” will cost some CPU time

Shit happens

- If someone steals PW hashes (and usernames and E-Mails) from your site: react quickly!
- Attacker might have passwords rather soon (depending on algorithm and pw quality).
- Will try them at your site (if useful) *and elsewhere*. Inform your users!
- Do a global password hash invalidation at your site(s) before the attacker can do real damage.

Password “resetting”

- P = INVALID_VALUE # never verifies!

  Note: setting the same super-secret long password for all users, then sending a password reset link is not a good idea – for all users *not* doing the reset: break one of them, break all of them!

- Notify users by E-Mail *and* other means:
  - send them a link to redefine their passwords,
  - never send users a password via E-Mail,
  - give an admin contact for trouble cases.

Password recovery insecurities

- Your mother's maiden name is not secret:
  - multiple sites you use might know the answer to that question – except if you **lied** differently to each of them.
  - for some people, one can even find the answer on wikipedia or somewhere else on the internet.
- As E-Mail is often used to reset a password, make sure your E-Mail account is rather safe or it might make you vulnerable everywhere else also.

Two-Factor Authentication

- First do a normal username / password check.
- If successful (and user has setup 2FA), let the user enter a 6 digit number generated by an app like Google Authenticator (RFC 4226) e.g. on a smartphone:
  - device and server share a random secret (can be easily shared via QR code)
  - secret is per account and per site
  - timed OTP only valid for 30s
- Libs for Python: pyotp, python-oath, …

Comments? Questions?