

TK2100: Informasjonssikkerhet

Lesson 11: SQLi and Passwords

Dr. Andrea Arcuri
Westerdals Oslo ACT
University of Luxembourg

Goals

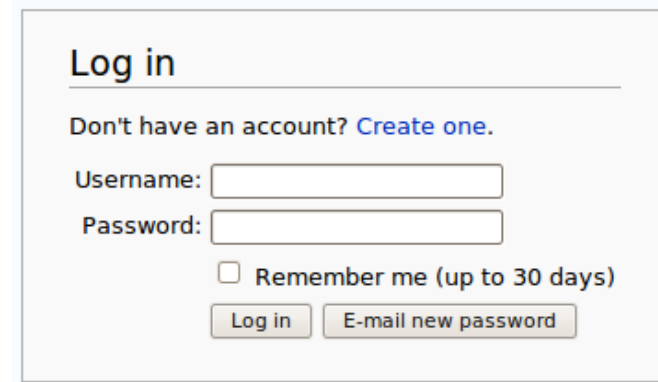
- Understand the bases of how SQL Injection is carried out
- Learn how passwords should be stored in a secure way

SQL Injection

Database Access

- Relational databases are the most common type of database
- Data can be queried with *SQL* (Structured Query Language)
- Typical SQL query: *SELECT * FROM x WHERE p*
 - *SELECT*: specify one, more or all columns/fields
 - *FROM*: the database tables from which data is read from
 - *WHERE*: a Boolean predicate to select just a subset of the data
 - i.e., only some specific rows, and not the whole tables
- SQL can be based on user inputs (eg HTML forms)

Example: Login



Log in

Don't have an account? [Create one.](#)

Username:

Password:

☐ Remember me (up to 30 days)

- “Users” table with “username” and “password”
- *String sql = “SELECT * FROM Users WHERE username=“ + username + “ and password=“ + password+”””;*
 - the variables “username” and “password” contain data from user inputs
- If query result is not empty, then the given user for that username is authenticated
- This works, but *can you see the problem here?*

SQL Injection: Tautologies

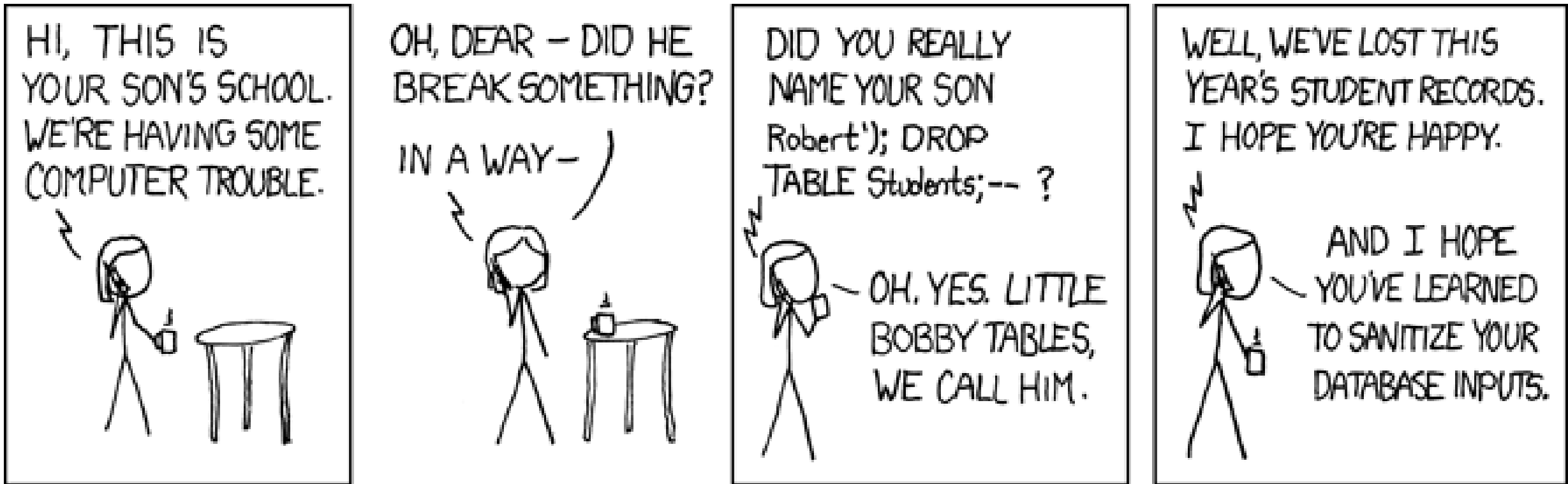
- Eve wants to login as Alice, but she does not know the password
- Not a problem! Can inject a tautology with the password, eg *“whatever’ or ‘1’=‘1’”*
 - note the first ‘ after “whatever”, and last string for 1 missing the closing ‘
- The resulting SQL would become: *“SELECT * FROM Users WHERE username=‘Alice’ and password=‘whatever’ or ‘1’=‘1’;”*
- The WHERE clause would always be true (ie, a tautology)

SQL Injection: Cont.

- Injection is the ***most common*** type of security vulnerability
 - See Top 10 from *www.owasp.org*
- Problem: creating SQL commands by concatenating string *without* sanitizing user inputs
 - can alter the structure of the SQL queries
 - non-escaped single quote ' was the problem in previous example
 - this is the same kind of issue as in XSS attacks
- SQL Injection allows attackers to access data without authorization by either turning the WHERE clauses into tautologies, or turn them in non-executed comments (--)

NEVER TRUST USER INPUTS!!!

Always Sanitize/Escape Inputs!



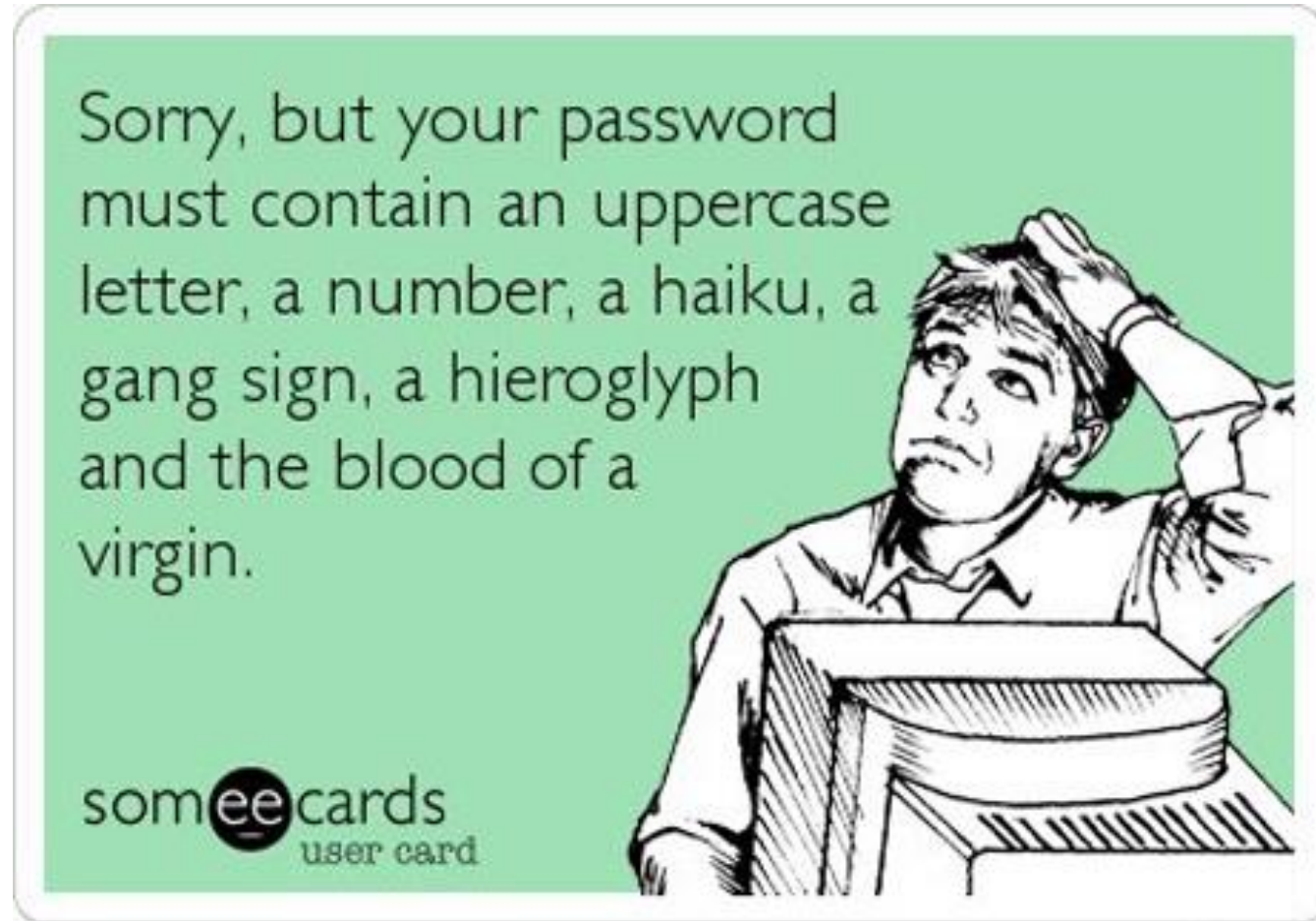
What info to steal?

- Any sensitive data... but most commons are:
- 1) Credit Card Numbers
- 2) Passwords
 - Hacker can then login an impersonate a specific user
 - People often re-use the same password for different web sites

Passwords

Passwords

- Needed to verify identity of a user
- Not too short nor simple, otherwise too easy to crack with brute-force
- *Security vs Usability*: hard to get a good balance
 - eg, ideally would have different passwords for each different site, and change them often, eg every week... but who the heck is going to do that???



Password Storage

- When creating new user, need to save password somewhere, usually a database
- **NEVER SAVE A PASSWORD IN CLEAR TEXT**
- Passwords need to be *hashed*
- Even if an hacker has full access to database, should **NOT** be able to get the passwords
 - Typical case is a successful SQL Injection attack
 - But many more cases: eg disgruntled employee, recovery from broken thrown away hard-drive, etc.
- Besides being able to impersonate a user, hacker can try the same password on other sites (Amazon/Facebook/etc)

- Thousands of examples...
- Eg, 2018, **1.3 million** plain-text passwords leaked from a web jewelry store
- On a different topic, look at the “shares” icons, and how such companies can keep track of me visiting that page...

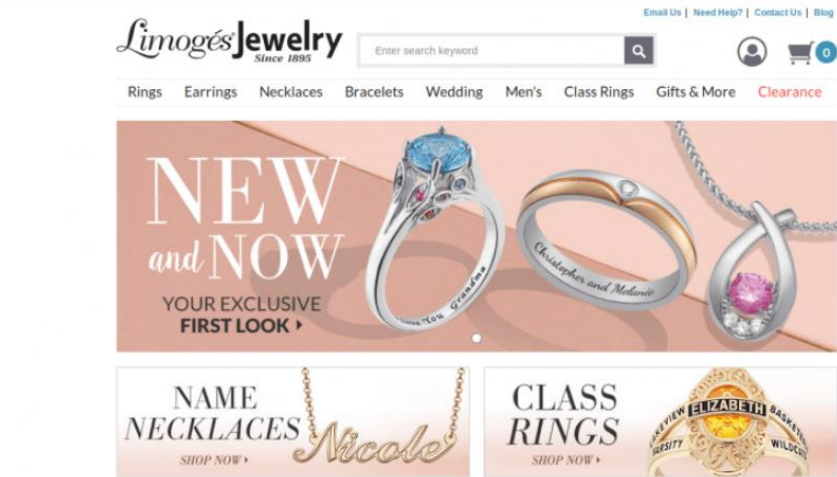
Secure | <https://thenextweb.com/security/2018/03/14/jewelry-site-accidentally-leaks-personal-details-plaintext-passwords-1-3m-users/>

News Conference New York Index TQ Deals Answers TNW X

LATEST INSIGHTS HARD FORK DISTRACT FULL STACK INVESTING 2.0

Jewelry site accidentally leaks personal details (and plaintext passwords!) of 1.3M users

by MATTHEW HUGHES — 5 weeks ago in SECURITY



Limogés Jewelry Since 1895

Enter search keyword

Rings Earrings Necklaces Bracelets Wedding Men's Class Rings Gifts & More Clearance

NEW and NOW YOUR EXCLUSIVE FIRST LOOK

NAME NECKLACES SHOP NOW

CLASS RINGS SHOP NOW

Best Sellers

339 SHARES

Facebook, RSS, Twitter, LinkedIn, Reddit, YouTube, Email

<https://thenextweb.com>

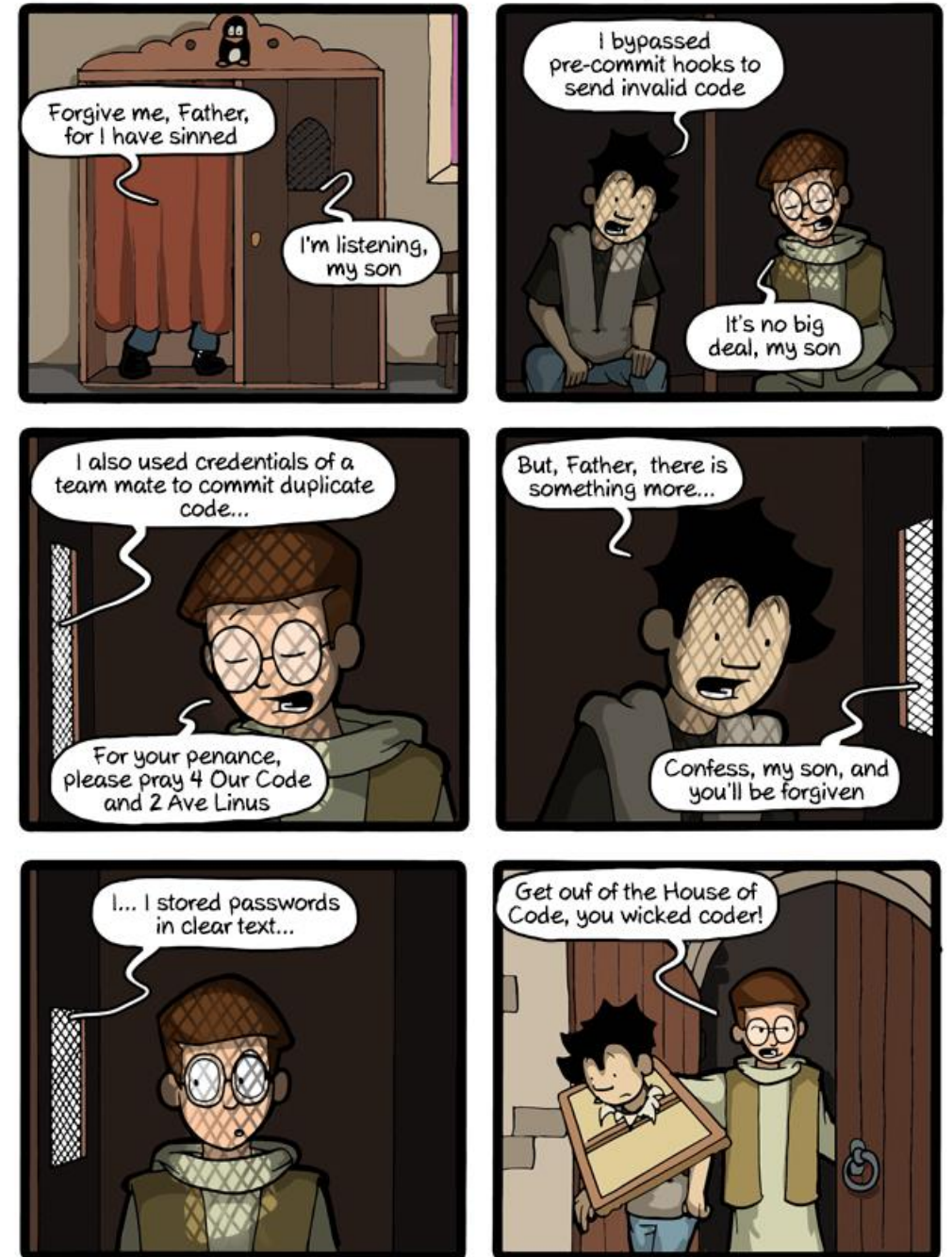
Few people are familiar with the Chicago-based MBM Company, Inc, but perhaps you might be familiar with its jewelry brand Limogés Jewelry. This firm sells cut-price trinkets through its website to customers across the US and Canada.

Researchers from German security firm [Kromtech Security](#) allege that until recently, MBM Company was [improperly handling customer details](#). On February 6, they identified an unsecured Amazon S3 storage bucket, containing a MSSQL database backup file.

According to Kromtech Security's head of communications, [Bob Diachenko](#), further analysis of the file revealed it held the personal information for over 1.3 million people. This includes addresses, zip-codes, e-mail addresses, and IP addresses. He also claims the database contained plaintext passwords — which is a big security

NEVER SAVE A PASSWORD IN CLEAR TEXT!!!

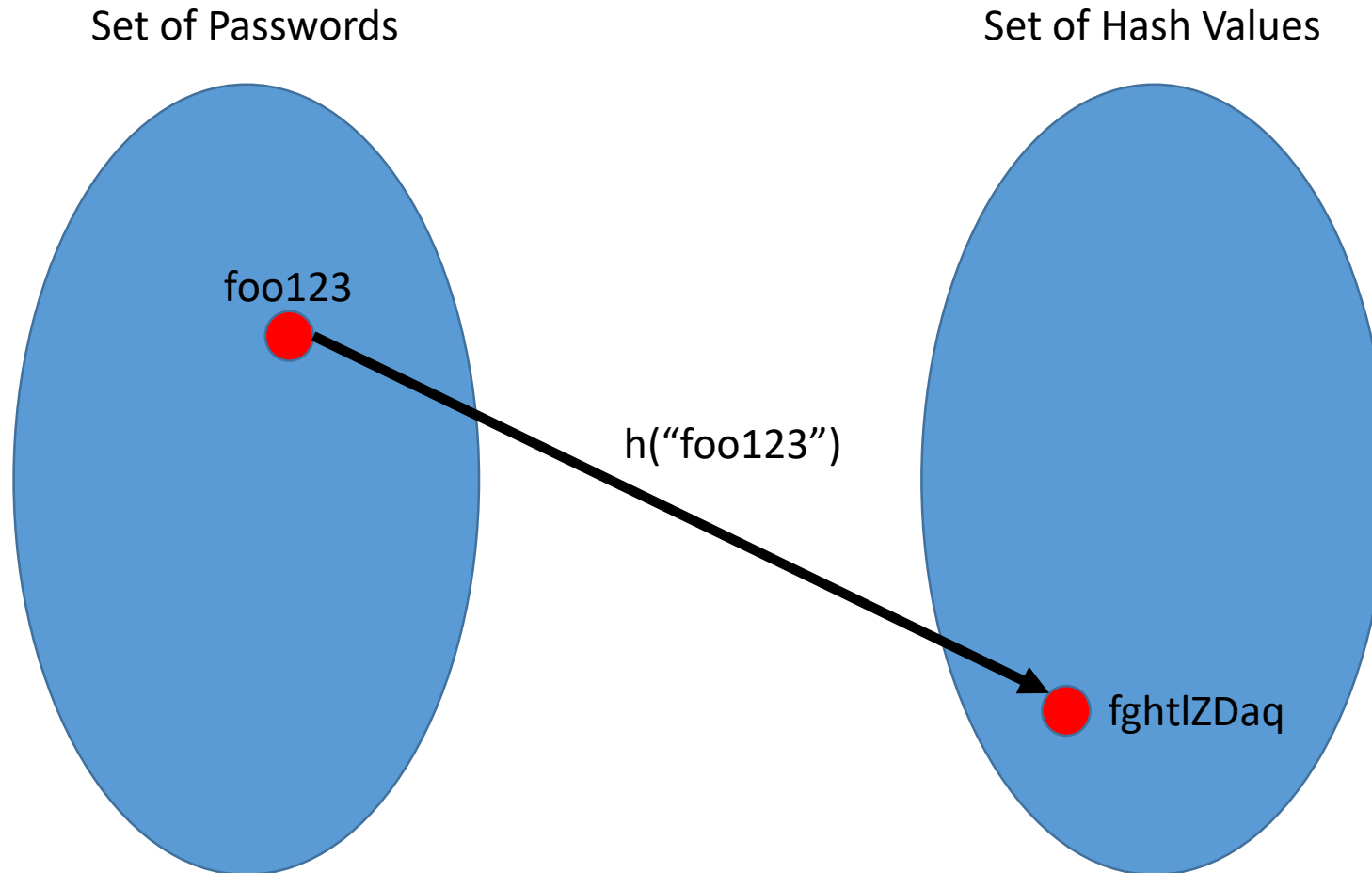
- If password is “foo123”, then such string should NEVER be saved in the database
- *But how to do authentication then?*



Hash Functions

- $h(x) = y$
- It is just a mathematical function from x to y
 - In our case, x is the password, and y is its hashed value
- Deterministic: always same y from same input x
- Shouldn't be able to recover x from y , even if you have full knowledge of how $h()$ is implemented
- Small change x' to x should lead to big different between y and y'
 - ie, y and y' should look uncorrelated, and so cannot say if x and x' are similar
- For security, want no collisions: no two values should have same hash, ie $h(x) = y = h(z)$

- $h(\text{"foo123"}) = \text{"fghtlZDaq"}$
- Should not be possible to recompute "foo123" if a hacker knows "fghtlZDaq"



Login with Hashed Passwords

- How can server verify the login of user A with password X, if the server does not know the password X, but only the hash $Y=h(X)$?
- Server needs to retrieve from database the hash Y for given user A, recompute the hash $h(X)$ from the input password X, and then verify that the new hash does match Y, ie $Y == h(X)$

Database table for Users

UserName	Hashed Password
...	...
foo	fghtlZDaq
...	...

- Logging in with *foo/whatever* would fail, because $h(\text{"whatever"}) \neq \text{"fghtlZDaq"}$
- Logging in with *foo/foo123* would work, as $h(\text{"foo123"}) == \text{"fghtlZDaq"}$

Salted Passwords

- Cannot expect users to have long passwords
- If hacker has access to DB, from a hash Y , can calculate $h(K)$ for all strings K up to certain length N , eg $N=8$, and check if any $h(K)$ does match Y
- For small N , this is doable. Do not even need to run $h()$, as those values can be pre-computed, ie *Rainbow Tables*
- Further issue: two users with same passwords will have same hash Y
- Solution: add a random *salt* S (eg a random long string) to the password before hashing, and store the salt together with the hash in the database
- $h(X+S)=Y$
- Each user will have its own random salt

Database table for Users

UserName	Hashed Password	Salt
...
foo	sfdsglll	SGFNSDSasdfs
bar	Hnjnvrs	dsfsfewaaa
...

- Both users *foo* and *bar* have the same password *foo123*
- The *hashed* values are different, because the *salt* is different
- $h(\text{"foo123SGFNSDSasdfs"}) == \text{"sfdsglll"}$
- $h(\text{"foo123dsfsfewaaa"}) == \text{"Hnjnvrs"}$

Pepper

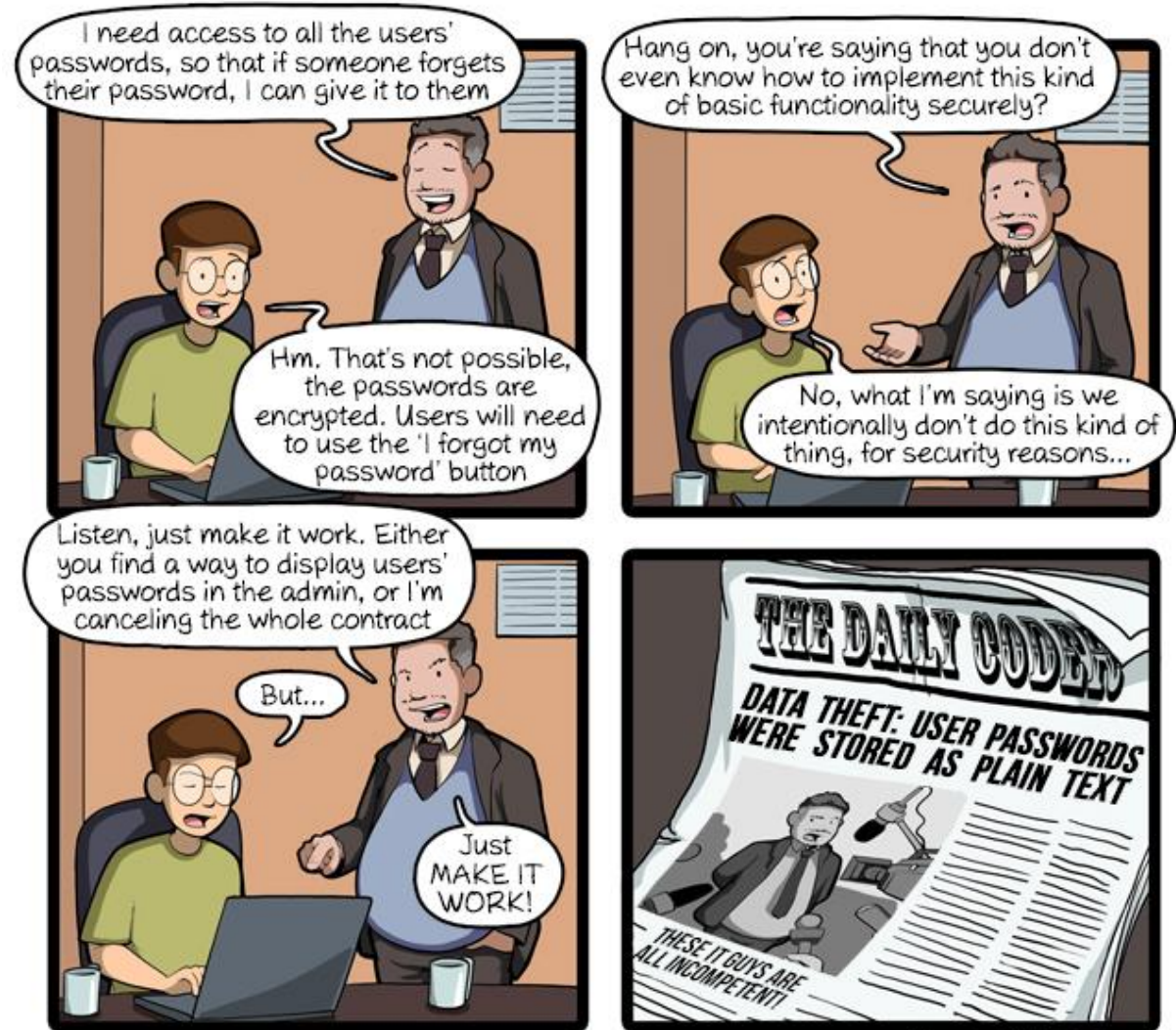
- If hacker has access to the database, s/he can read the *salt* values
- Still non-trivial to break the hash code, but doable
- *Pepper*: yet another random string added before calculating the hash
- NOT stored in the database, just somewhere else
 - files, remote server, hardcoded in the source code, etc
- One single pepper string for whole application (and not per user)
- Help mitigating if hacker gets access to the database (eg via SQL Injection), as would not be able to read the pepper

Hash Function Speed

- In security, you want hash functions that are *slow*, to make it difficult for the hackers to break them
 - but still manageable time on server to do authentication, eg within 1 second
- *BCrypt* is the most used hash function for passwords
- However, you can make *slow* any hash function (eg SHA256) by using a loop, in which the output is re-hashed N times
 - eg, $N=6$ and so $h(h(h(h(h(h(x)))))) = y$
 - Note, SHA stands for Secure Hash Algorithm, but it is not really secure on its own, as too fast

Implications of Hashing

- A web site should not be able to tell you what was your password
- If can give it to you, or even just some hints (eg first 2 letters), or tell you if too similar to a previous password, then *BE WORRIED...*
- Note: comic has a mistake... passwords must be hashed, and NOT encrypted

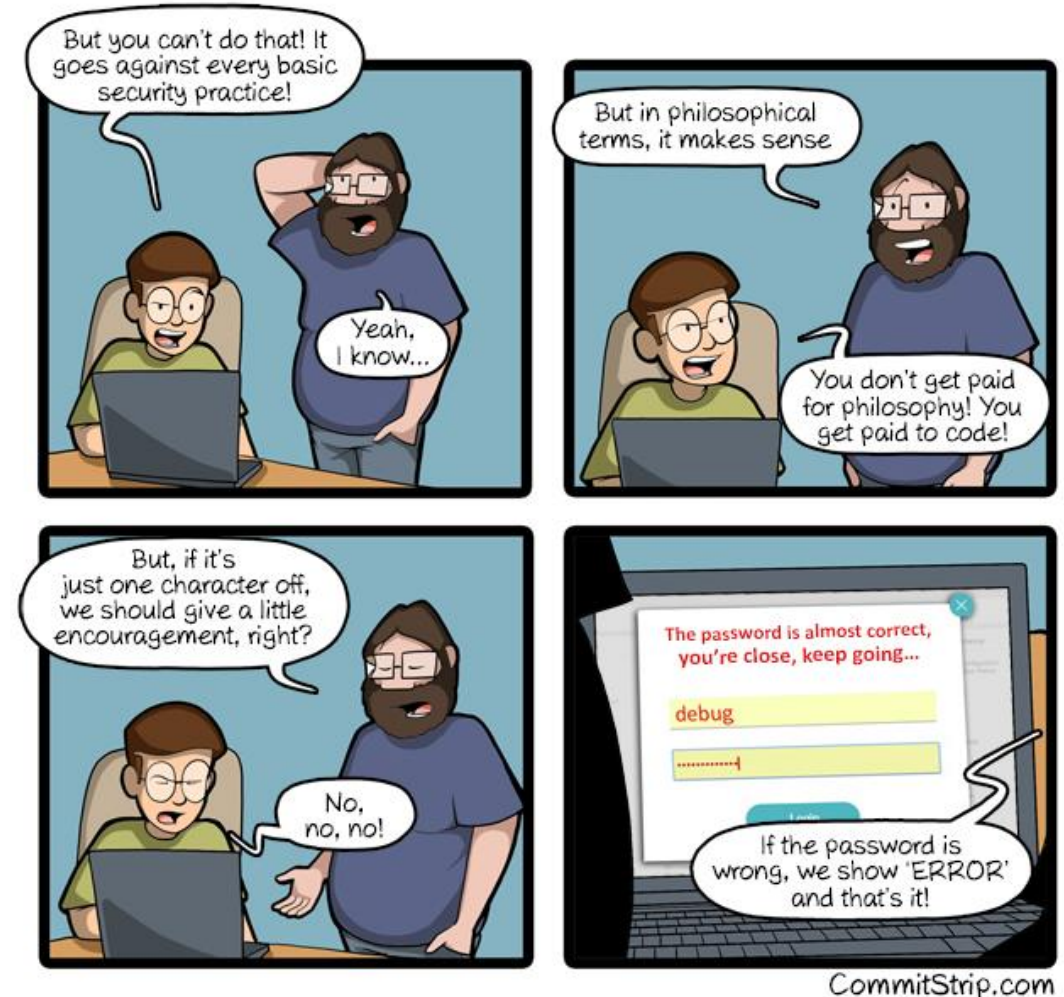


Password Recovery

- Should **NOT** be possible to recover a password
- If one forgets it, can have a password *reset*
- Email sent to user, with link to create a new password
- This is a typical approach, but only works as long as email is not compromised

Failed Login

- A failed login should not provide any further info
- Eg, should not tell if password is close to correct one, or if user even exists
- No unnecessary info should be leaked, as *hacker could use it*
- Example: if user does not exist, still need to compute hash for password... *why???*

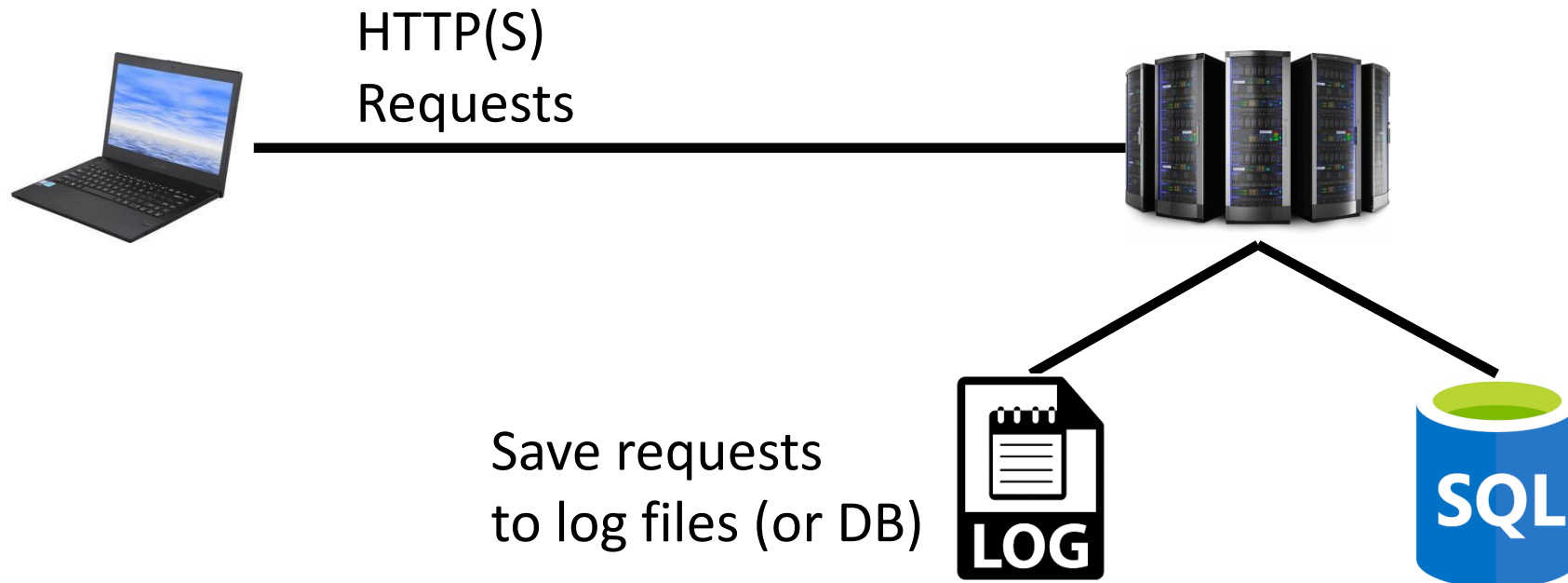


Cont.

- Assume login with *foo/foo123*
- Assume user *foo* does not exist
- Computing hash $h(\text{"foo123"})$ takes time, so one could just immediately state that login did fail, as *foo* not in the database
- Computing $h(\text{"foo123"})$ would be a waste of CPU time, but still necessary
 - Recall, $h()$ is slow
- If not computing $h()$, then login for non-existing users would be much *faster*, and hacker can use such info to determine if a user exists
 - i.e, check how long it takes for server to state failed login
 - i.e., leaked info based on response time of the HTTP requests

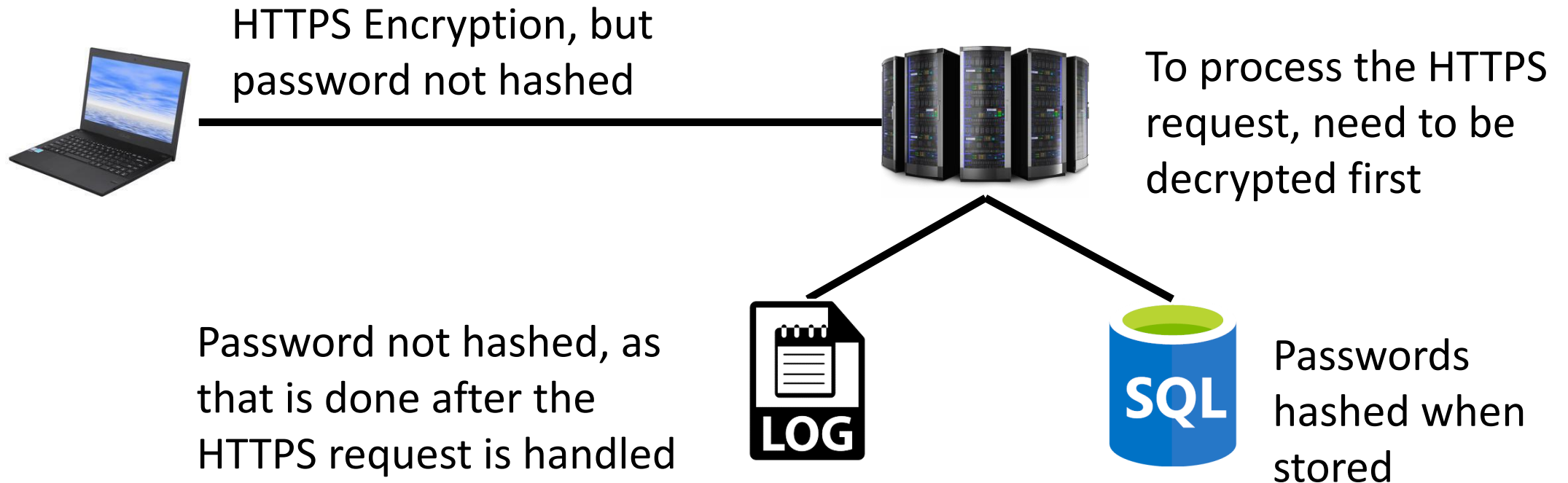
Logging HTTP(S) Requests

- Not uncommon to “log” events in a web server
 - eg, for debugging purposes, to see how users interact with the web application
- *What is the “security” problem here?*

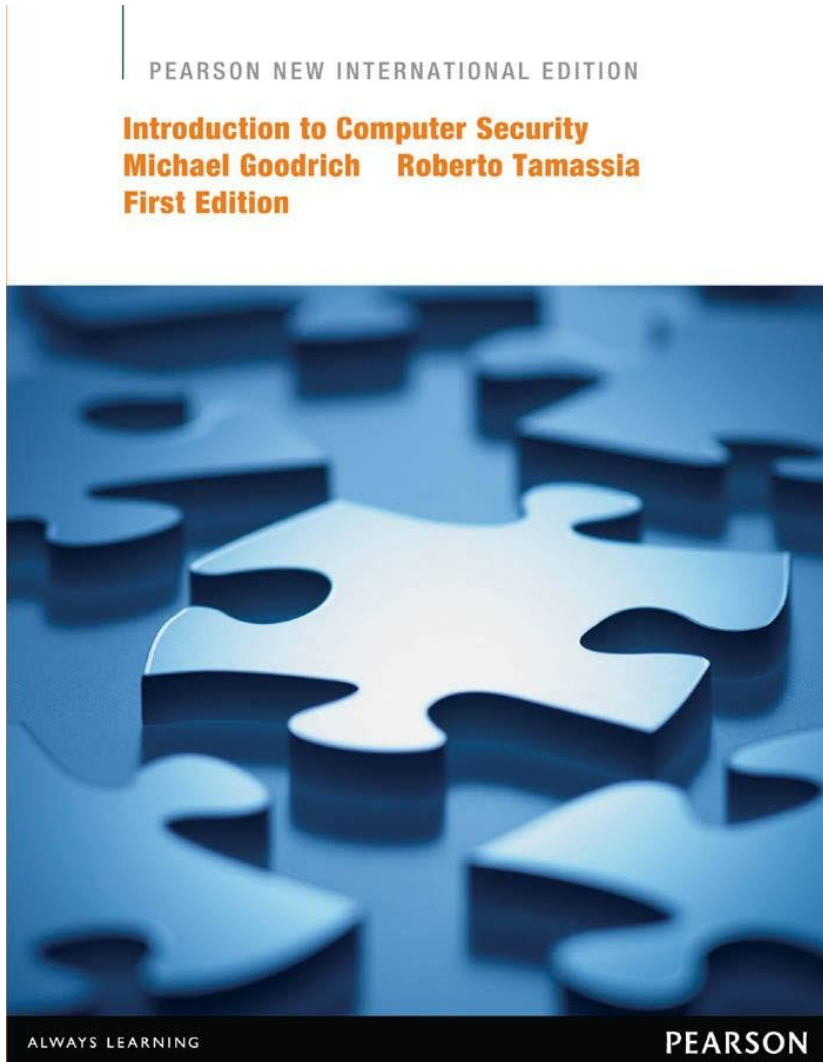


Passwords in logs not hashed!!!

- 2018: actually happened at *GitHub* and *Twitter*
- When doing logging of HTTP requests, should remove/filter out data of password



For Next Week



- Book pages: 41-42, 137-138, 372-377, 417
- Note: when I tell you to **study** some specific pages in the book, it would be good if you also *read* the other pages in the same chapter at least once