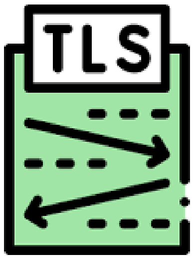
Security II
Password-based
user authentication

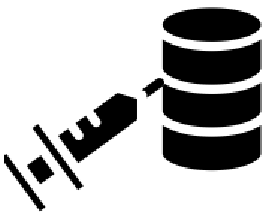
Monday Nov. 10th, 8.15-12.00
(assignment work 12.15-16.00)

Niels Jørgensen

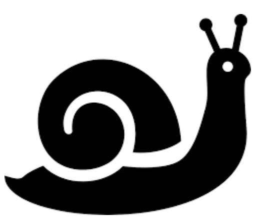
Three security course days

1. Secure Socets + SQL Injection
   - SSL intro: symmetric + asymmetric encryption
   - How to prevent injection of malign SQL queries via our user-interface?

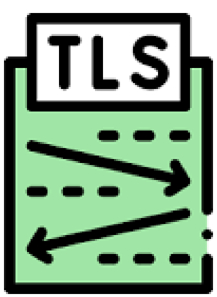




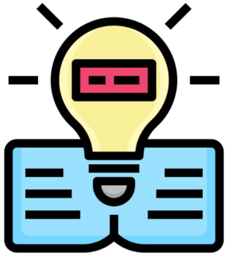
2. Password-based authentication
   - Hashing of users' passwords
   - How to slow down brute-force attacks but retain fast login of legitimate users?



3. Secure sockets (SSL) (revisited)
   - "Forward secrecy"



Approach:
theory + hands-on



A theoretical topic
- SSL

Two practical, hands-on topics
- SQL injection
- Password hashing
- you will be asked to make modifications
  to programs downloaded from moodle
- password program may be used in your project

Exam questions (suggested, not final)

Why should passwords be hashed?

Why should passwords be salted?

Why should the password hash function be iterative?

Today's plan

 SQL injection wrap-up

Introducing "PasswordbasedAuthenticator"

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative

Discussion

Assignment 5
due today 23.55

Questions 1-3 are about SQL Injection.

Question 1. Define a stored database function that returns a course from the course table in the university database. You may name the stored function safe_course(). The function should have one input parameter of type VARCHAR(8). The function should return courses whose course id match the input. (There will be at most one such course, since the course id is a primary key of the table). Naturally, the function should leave out any course offered by the Biology department. Show the SQL code that defines the function.

Question 2. In SQL-Injection-Frontend, in the switch statement of Program.cs, define an option ‘sc’ that calls a method safeComposedQuery(). Also, in QueryConstructor.cs, define method safeComposedQuery(). The new function should be an improvement over composedQuery() in two ways: firstly, it must call the stored database function safe_course(); and secondly, it must use separate parameter passing, including by calling query/3 as defined in PostgreSQL_client.cs.

Question 3. Define an SQL injection attack that works when option ‘c’ is selected, but fails when option ‘sc’ is selected. Provide a screenshot of the succesfull attack and a screenshot of the failed attack.

in safeComposedQuery(): firstly: call safe_course

```
string staticSQLbefore = "select * from course where course_id = '";
Console.Write("Please type id of a course, or part of id: ");
string? user_defined = Console.ReadLine();
string staticSQLafter = "' and dept_name != 'Biology'";
string sql = staticSQLbefore + user_defined + staticSQLafter;
```

```
string staticSQLbefore = "select * from safe_course(‘”;
Console.Write("Please type id of a course, or part of id: ");
string? user_defined = Console.ReadLine();
string staticSQLafter = “’)";
string sql = staticSQLbefore + user_defined + staticSQLafter;
```

safeComposedQuery()
Firstly: .. still not safe

```
Please type id of a course: BIO-101') ; select * from course; --
Query to be executed: select * from safe_course('BIO-101') ; select * from course; --')

 course_id | title | dept_name | credits
-----------+-------+-----------+---------
(0 rows)

 course_id | title                      | dept_name   | credits
-----------+----------------------------+-------------+---------
 BIO-101   | Intro. to Biology          | Biology     | 4
 BIO-301   | Genetics                   | Biology     | 4
 BIO-399   | Computational Biology      | Biology     | 3
 CS-101    | Intro. to Computer Science | Comp. Sci.  | 4
 CS-190    | Game Design                | Comp. Sci.  | 4
 CS-315    | Robotics                   | Comp. Sci.  | 3
 CS-319    | Image Processing           | Comp. Sci.  | 3
 CS-347    | Database System Concepts   | Comp. Sci.  | 3
 EE-181    | Intro. to Digital Systems  | Elec. Eng.  | 3
 FIN-201   | Investment Banking         | Finance     | 3
 HIS-351   | World History              | History     | 3
 MU-199    | Music Video Production     | Music       | 3
 PHY-101   | Physical Principles        | Physics     | 4
(13 rows)
```

safeComposedQuery()
.. still not safe (cont.)

```
Please type id of a course: BIO-101') ; select * from course; --
Query to be executed: select * from safe_course('BIO-101') ; select * from course; --')
```

user-defined input: BIO-101') ; select * from course; --
passed to safe_course(): 'BIO-101'

So even though safe_course() ..
- does some parameter checking
- does the censoring (not allowing Biology courses)

.. we are still hacked

in safeComposedQuery(): secondly, use query/3 for separate parameter passing

public void query(string? sql, string? name, string? val)

“select * from safe_course(@course_id)“

(where @course-id is a parameter name)

@course_id
the parameter name

CS-101
the parameter value
could be really malicious BIO-101’) ; select * from course; --
but now definitely will be understod as parameter

Using the library Npgsql to connect to the database

The C# program SQL-Injection-Frontend.cs
firstly sets up a database connection,
which is represented by an instance of class NpgsqlConnection:

```
String s = "Host=localhost;Username= ..";
NpgsqlConnection con = new NpgsqlConnection(s);
con.Open();
```

Using Npgsql to execute SQL commands (without separate parameter passing)

The C# program uses an instance of class NpgsqlCommand to represent the SQL statement

```
public void query(string? sql) {
 ..
 NpgsqlCommand cmd = new NpgsqlCommand(sql, con);
 .. cmd.ExecuteReader();
```

con is the database connection

query/3 uses a Npgsql feature for separate parameter passing

```
public void query(string? sql, string? name, string? val) {
   ..
   NpgsqlCommand cmd = new NpgsqlCommand(sql, con);
   cmd.Parameters.Add(new NpgsqlParameter(name, val));
   .. cmd.ExecuteReader();
```

the command
(or query) cmd
is extended with
the parameter
name/value pair

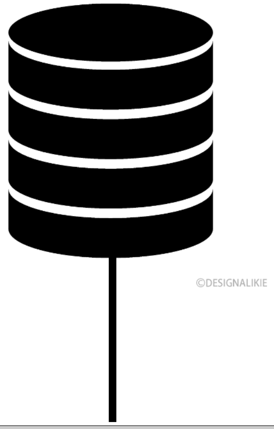
Summary of separate parameter passing

select * from safe_course(@course_id)

name: @course_id
value: “CS-101’); select * from course; --”

this malign value
will be identified as invalid
because it is *passed*
separately as a parameter

in addition, remainder of
safe_course() definition
with "and course.dept_name != 'Biology' ; "
is always executed
(never ‘commented out’)
because it is part of code of stored function



create or replace function safe_course(c_id varchar(8))
...

Summary (cont.)

These recommendations are from OWASP Foundation
- Open Web Application Security Project
- URL
  - cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
  - (accessed Nov 4, 2025)

Primary Defenses:

- **Option 1: Use of Prepared Statements (with Parameterized Queries)**
- **Option 2: Use of Properly Constructed Stored Procedures**
- **Option 3: Allow-list Input Validation**

- we can do prepared statements in PostgreSQL as well
  - of course with separate parameter passing
- ‘properly constructed’ (??) - perhaps they mean with separate par. pass.
- separate parameter passing is not a 100% guarantee

Exam questions - SQL injection

An example malicious SQL injection

Protection against SQL injection
using stored functions

Explain how to define an appropriate *stored function*

Emphasize separate parameter passing in *calls* to stored function

Implementation of separate parameter passing by query/3
- query/3 distinguishes between parameter names and values
- parameters names should begin with @, ie., @course_id
- query/3 calls function Parameters.Add of class NpgsqlCommand

Competion

What is the shortest SQL-injection hack
you can come up with
that displays at least one Biology course?

Hack from my notes uses 11 characters.

```
Please type id of a course: BIO-101' --
Query to be executed: select * from course where course_id = 'BIO-101' --' and dept_name != 'Biology'

 course_id | title              | dept_name | credits
-----------+--------------------+-----------+---------
 BIO-101   | Intro. to Biology  | Biology   | 4
(1 row)
```

Today’s plan

(SQL injection wrap-up)

Introducing “PasswordbasedAuthenticator”

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative

Discussion

Learning goals

.. to understand purpose and means of defenses

- eg., purpose of hashing + how hashing meets purpose

.. to assess the degree to which purposes are met

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative

Assignment 5 (cont.)

Questions 4-6 are about passwords.

Question 4. Implement a check that the password provided by a new user at registration contains more than eight characters. Also check that the password does not contain the username. Obviously, the password ‘admindnc’ for username ‘admin’ will fail this test. Checks should be done in the definition of method passwordIsOK() in Authenticator.cs. [Then there is a comment about passwordIsOK()]

Question 5. Implement iterative hashing. Hint: Modify the C# source file Hashing.cs. What number of iterations appears to be reasonable on your computer?

Question 6 is about passwords *and* SQL injection.

Question 6. In Authenticator.cs, the method sqlSetUserRecord() defines a string that is an SQL command. The SQL command is then used in method register(). Is the method vulnerable to SQL injection? If your answer is ‘yes’, provide an example of a successful injection. If your answer is ‘no’, provide an example of a failed injection attempt. *If you believe the method is vulnerable, a good solution should include a screenshot that documents an attack.*

PasswordBasedAuthenticator is ..

.. a prototype
- for teaching
- some parts may be useful in your projects

.. with a C# part:
- download three C# source files into new project directory
- add relevant package to directory
- (for C# connection to PostgreSQL)

.. and a database part:
- you must define database ‘passwords’
- with one table
- if useful for your project,
  table *may* be moved to your existing project database

Table password

```
create table if not exists password (
  username varchar(50),
  salt char(16),
  hashed_password char(64),
  primary key (username)
);
```

username is defined as primary key

- simplifies design
- case ‘username already taken’ -> violates primary key constraint
- in mature application, perhaps identify user by a user ID (number)
- (and of course then update/extend the C# interface to the database)

Perhaps you are already using table password in your projects?

User interface (end user) in C#

```
Please select character + enter
'r' (register)
'l' (login)
'x' (exit)
>
```

Program.cs

Defines end user interface
calls register() and login()

Authenticator.cs

Defines login() and register()
connects to database
calls hash()

Hashing.cs

Defines hash()
generates salts

User interface (developer, administrator) in SQL

```
passwords=# select * from password;
 username |       salt       |                          hashed_password
----------+------------------+------------------------------------------------------------------
 admin    | ED5D50781D89EF20 | 4DC128AB9402D6DAE7F0EE718A8F37E8D6410722BBAF7EE5AABA6A6029B4FFF4
(1 row)
```

The admin-record ..
- may be added from the end user interface
- register(admin, admindnc)

Email server of Democratic National Committee (DNC) hacked (2016)
- 27,000 emails leaked
- revealed bias in favor of Hillary Clinton, against Bernie Sanders
- careless system administrator?
- or perhaps the hackers used SQL Injection??

Bits, bitstrings: storage + display

```
passwords=# select * from password;
 username |       salt       |                         hashed_password
----------+------------------+------------------------------------------------------------------
 admin    | ED5D50781D89EF20 | 4DC128AB9402D6DAE7F0EE718A8F37E8D6410722BBAF7EE5AABA6A6029B4FFF4
(1 row)
```

salt ED5D..

- 64 bits

hash 4DC1..

- 256 bits

Hexadecimals

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- good compomise for a prototype (maybe also your project)
- storing salt/hash as hex text (reasonably small, space efficient)
- display witthout conversion (human readable)

Exercise (hexadecimals)

(A) How many different numbers (salts) can you store using 16 “hexes” (hexadecimal digits)? Show result in decimals.

(B) What is the memory penalty for storing salts as hexes instead of using a low-level bitstring format?

Penalty = the extra space required for storing one salt

Asssume -

- Postgres has a low-level bitstring format that uses only one byte for eight bit
- Postgres stores a character in one byte

Today’s plan

(SQL injection wrap-up)

Introducing “PasswordbasedAuthenticator”

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

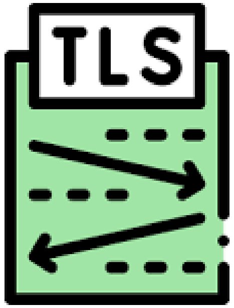
Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative

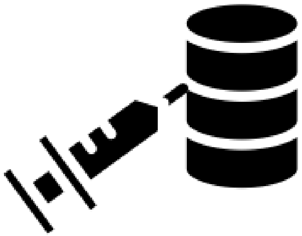
Discussion

Which are the most important threats?

Threats to break encryption?
(such as SSL encryption)
- ?



Threats to inject macilious code?
(such as in SQL injection)
- ?



Threats to guess user’s passwords?
- ?

Why use passwords?

Let us ask (as in the TRIN model from humtek):

1. what is the purpose of using passwords?
2. how do passwords achieve this purpose?

Question 1:

Purpose is to support user authentication

User authentication = verification of a claimed identity

login(admin, admindnc)

claimed identity

verification

Exercise
(password weaknesses, alternatives)

(A) In general terms, what are the sources of the weaknesses or risks associated with password-based user authentication?

(B) Also in general terms, what are the alternatives?

Defenses #1-#4 date back to the late 1970s

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative

See Morris and Thomsen (1979)

Defense #1: Passwords must have some minimal length

Danish Council for Digital Security
(Rådet for Digital Sikkerhed, https://www.digitalsikkerhed.dk/)

- Password length must be above a lower limit: at least eight characters
- At login, when a user provides a wrong password, there must be a time delay before a new password can be entered
- Also at login, there must be an upper limit on the number of wrong passwords entered
- Also recommends .. checking against very weak passwords

PasswordBasedAuthenticator: further assumptions

The assumptions have not been implemented
- used in attack analysis

System assumptions:

- passwords must be at least eight characters
- upper limit of ten wrong passwords
- a total of 50,000 user accounts
- half of user accounts' passwords are exactly eight characters
- think of RUC or an even larger university
  - though a uni that allows passwords > 8 characters
  - but does not require 'complex passwords'

Exercise
(password recommendations)

(A) What password requirements would be appropriate
in your project?

(B) In particular, would you require or recommend complex passwords?
Complex =
- lower case + upper case
- characteres + digits
- special characters, such as %&/(

Or start working on assignment question 4 (password check)

Attack types

Online attacks
- In an online attack, the attacker attemps to log in with some username and a guessed password. If the attempts fails, the attacker tries another password, and so on.
- Defense #1 provides (some) protection against online attacks

Offline attacks
- In an offline attack, the attacker has a <u>copy of the password table</u>. That is, the attacker has a copy of every password record, including the username and the stored copy of the password (probably hashed)
- Defenses #2-#4 provide (some) protection against offline atttacks

Also, there are *social engineering* attacks
- 'email phishing' as in Celebgate (2014) (obtained passwords of celebrities)
- guidelines may provide (some) protection against social engineering

Password dictionary attack (attack #1)

Type: online
Target: any user account (not a particular account)
Method: intelligent password guessing using a password dictionary

A password dictionary is a collection of likely passwords
pizza
pasta
admindnc
..

Password dictionary attack uses ..

Intelligent password guessing as opposed to:
- brute force password guessing
- that is, trying all possible character combinations

Total number of possible combinations (eight characters): 95^8
- because there are 95 printable characters
- $95^8 = 6,634,204,312,890,625$
- this is approximately 2^{52}

Thus in a brute force, online attack,
accounts will be closed before a successful guess.

A password dictionary may be built from ..

- previously leaked passwords
- English dictionaries
- samples of ordinary English texts

Example password dictionary (very large)

- http://crackstation.net

Password dictionary attack (implementation)

Implementation (pseudocode)

```
for i=1 to 10 do {
    for each username u of the total of 50,000 usernames do {
        select a random eight character password p from the pwd dictionary
        try login(u, p)
    }
}
```

Dictionary attack: likelihood of success

Password entropy

Example

- an eight character password (assumed to be normal, not complex) has “entropy of 18 bits”

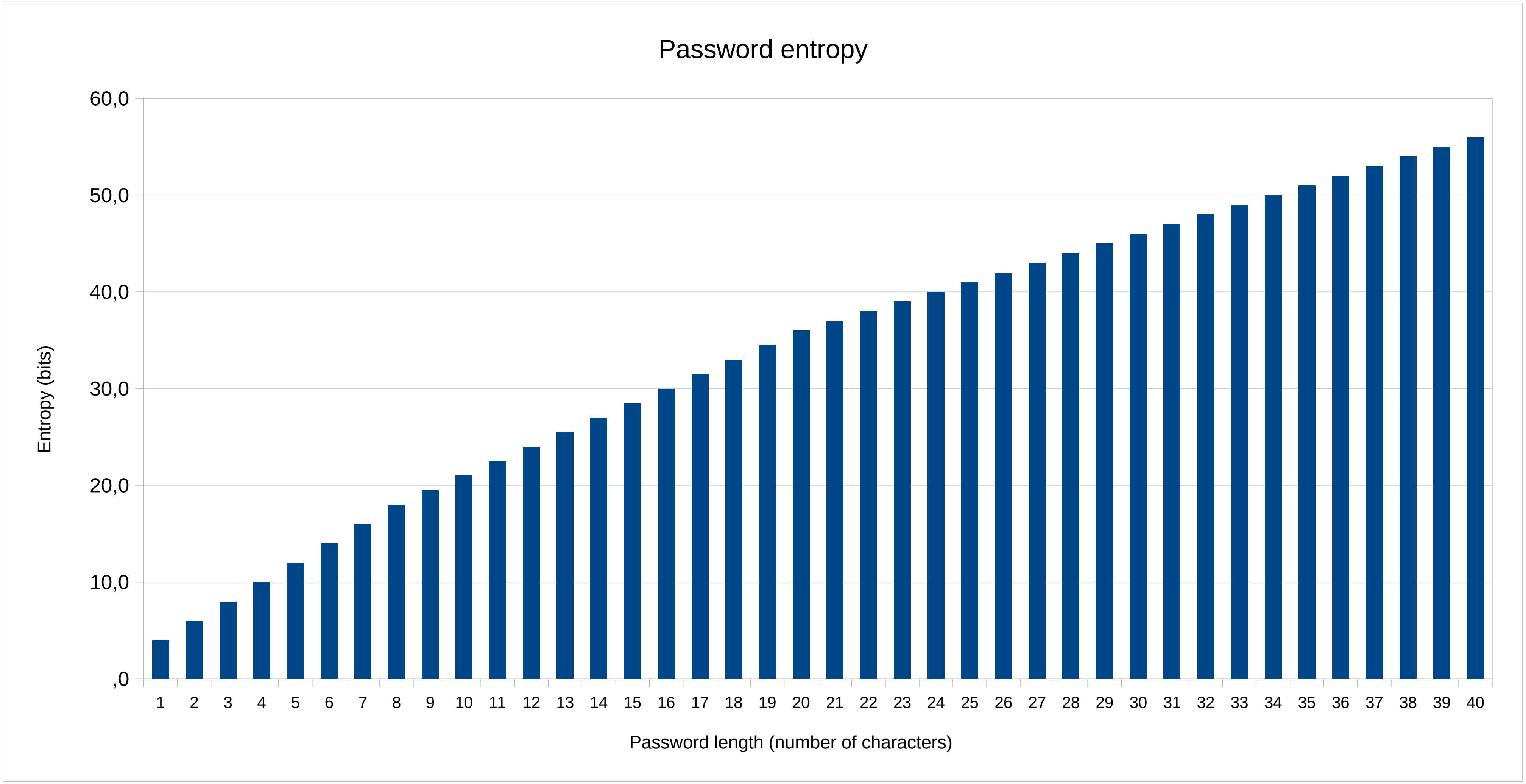
Meaning

- “the number of guesses required to crack a password with entropy n = same number required to guess an arbitrary string of n bits”
- guessing an arbitrary n-bit string requires 2^n guesses (on the average you succeed after half the guesses)

The example is based on an analysis from NIST (National Institute of Standards and Technology) which assumes

- passwords have same randomness as ordinary English text
- “pizza”, “pasta”, .. are likely passwords
- “admindnc” somewhat likely
- &3(/ax;} is extremely unlikely

Password entropy



Password dictionary attack (likelihood of success) (cont.)

2^{18} = 262,144 or approximately 250,000

We also assume
- attacker has password dictionary and picks guesses randomly
- likelyhood of success of one guess (if account password is eight chars) = 1 / 250,000 = 0.000,004 = 0.000,4%

Likelyhood of success of ten guesses on same account
- approximately 10 * 0.000,4% = 0.004%

Likelyhood of success of guessing some password (out of 25,000)
- approximately 25,000 * 0.004% = 10%

Password entropy exercise

Let’s assume that 25.000 user accounts
now have 12 character passwords (instead of 8 as before).
(Then entropy is approx. 2^{24}.)

What is the likelyhood of success of a password dictionary attack
targetting some account among those 25.000 accounts?

Summary

Attack#	Attack name	Type	Target	Defenses	Conclusion
Attack #1	Passwod dictionary attack	Online	Some account	Password minimal length, “ten guesses, you’re out”	A real risk
Attack #2					
Attack #3					

As an administrator, you may suggest stronger password requirements

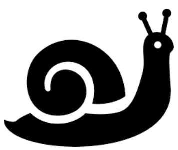
As an individual, you may consider choosing a stronger password

Recall:
- defenses #2-#4 do not protect against online attacks
- many systems use “MFA” = multi-factor authentication

Passwords - today's plan

Introducing "PasswordbasedAuthenticator"

Defense #1: Passwords must have some minimal length



Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative

Discussion

Hashed passwords

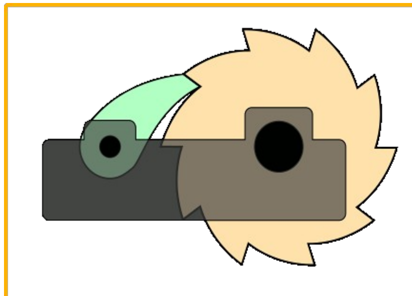
Passwords should be hashed using ..
- .. a strong cryptographic hash function

Crucial property: such a function is a ‘one way-function’

sha256
- the hash function used in PasswordBasedAuthenticator

Example
- password (hash input): admindnc
- hashoutput: **B94EAB632F442F573DED7C5902DC12EB 9A2DB87046F39106106CF2A3B90A96D3**
  (64 hexes, 256 bits)
- check it out: https://miraclesalad.com/webtools/sha256.php

‘one way-function’ property
- if you know the password, the hash is easy to compute
- if you know the hash (only), you can not infer the password



Why hash passwords?

Passwords must be hashed
- because of the risk that the password table is leaked
- ie., the record (admin, admindnc) is leaked to the attacker
- obviously this defeats the purpose of using passwords

Password leak examples
- Morris and Thomsen (1979) case: all passwords published
- Many social media cases (see Wikipedia)
- Our system administrators should not be 100% trusted

Therefore, we should design the password table to be
1. passwords are protected even if table is leaked
2. still consider password table highly confidential
- restrict table access for people and programs

Strong cryptographic hash functions

A strong cryptographic hash function *h* must satisfy:

1. The input *p* to *h* may be of any size.
2. The output *h(p)* has a fixed, small size.
3. Given an input *p*, it is easy to compute *h(p)*.
4. Given an output *v* = *h(p)*, it is practically impossible to find *p*.
5. There is no known pair of values *p* and *p'* such that *h(p)* = *h(p')*

Item 1: permits arbitrary length of passwords

- we may add salts (defense #3)

Exercise:
(A) In what way do items 1, 2 og 5 imply a contradiction?
(B) How is it “resolved” ?
(C) Instead of hashing passwords, why not encrypt passwords,
and then use encryption key to decrypt them when needed?
“When needed” = function login() would decrypt to compare wih user’s passw

Passwords - today's plan

Introducing “PasswordbasedAuthenticator”

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative

Discussion

A password table with ‘salted passwords’

```
passwords=# select * from password;
 username |       salt       |                             hashed_password
----------+------------------+------------------------------------------------------------------
 admin    | ED5D50781D89EF20 | 4DC128AB9402D6DAE7F0EE718A8F37E8D6410722BBAF7EE5AABA6A6029B4FFF4
(1 row)
```

PasswordBasedAuthenticator already implements defense #3

Thus, in the above record, the field ‘hashed_password’ ..
- is not sha256(admindnc)
- rather it is sha256(admindnc + salt)
- salt is ED5D.. (randomly generated)
- and stored in the password record
- so the salt is public in the sense that password table is semi-public (thas is, protected - but not 100% secret)

Why salting?

```
passwords=# select * from password;
 username |       salt       |                         hashed_password
----------+------------------+------------------------------------------------------------------
 admin    | ED5D50781D89EF20 | 4DC128AB9402D6DAE7F0EE718A8F37E8D6410722BBAF7EE5AABA6A6029B4FFF4
(1 row)
```

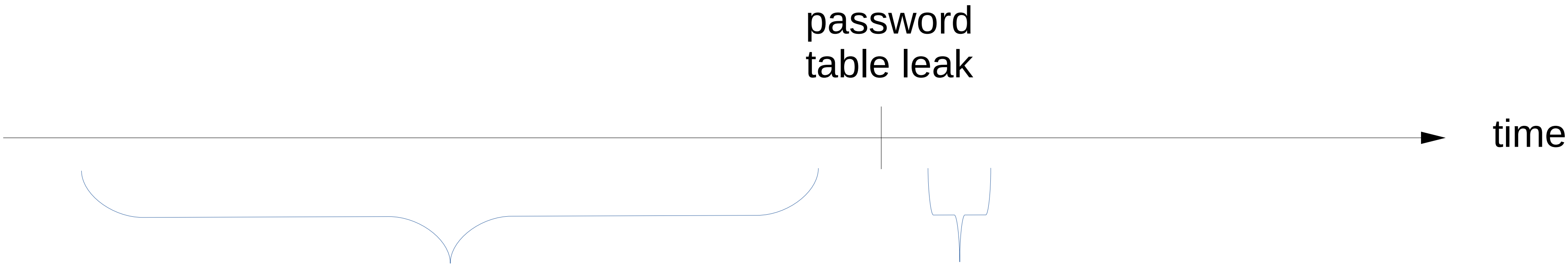
Suppose our password table stores only pairs (username, hashed_password),

ie., (admin, B94E ..)

Then our system would be vulnerable to a *rainbow table attack*

- a kind of offline attack
- where attacker has a table of hashvalues of common passwords
- such as the hash value of ‘admindnc’

Ranbow table attack: building table in advance of password table leak



Build table in advance of leak
- 99% of work
- rainbow table is independent of actual password table

Do table look-ups
- 1% of work

How is a salt generated?

```
byte[] salt = new byte[salt_bytesize];
rand.GetBytes(salt);
```

In the above code from Hashing.cs

- variable rand points to a ‘Pseudo-random number generator’
- which provides 64 bits that are ‘pseudo-random’

The salt should be *unpredictable* for the attacker

- if attacker knows username admin has salt 1
- and username peter has salt 2
- etc.
- the salt would not serve to make it substantially more difficult for an attacker to build rainbow table

Pseudo-random number generators

A random number is
• a number that is *unpredictable*

‘Unpredictable’ even if the attacker knows ..
.. how the pseudo-number random generator works
.. all the previous (pseudo-random) numbers

A pseudo-random number generator (PRNG)
is *cryptographically secure*
• if there is no known method for predicting the next number / salt

Cryptographically secure PRNGs are ..
• *vital* for generation of encryption *keys*
• *desirable (not vital)* for genetation of password salts
• I am aware of no password attacks that utilize ‘weak’ PRGNs

How is a salt ‘added’?

```
private string hashSHA256(string password, string saltstring) {
  byte[] hashinput = Encoding.UTF8.GetBytes(saltstring + password);
  byte[] hashoutput = sha256.ComputeHash(hashinput);
  return Convert.ToHexString(hashoutput);
}
```

In the above code from Hashing.cs

- saltstring and password (as texts) are added
- salt first, then password
- then converted to a byte-array
- then hashed
- (and finally converted to a string of hexadecimals)

Rainbow table attack (attack #2)

Type: offline
Target: any user account (not a particular account)

Method: intelligent password guessing using a rainbow table

A rainbow table is a table of pairs (password, password_hash)

(pizza, hash(pizza))
(pasta, hash(pasta))
(admindnc, hash(admindnc))
..

Rainbow table attack: exercise

Type: offline
Target: any user account

Method: intelligent password guessing using a rainbow table

A rainbow table is a table of pairs (password, password_hash)

(pizza, hash(pizza))
(pasta, hash(pasta))
(admindnc, hash(admindnc))
..

Exercise:
What would you use as a source of likely passwords for a rainbow table?

Rainbow table attack: implementation

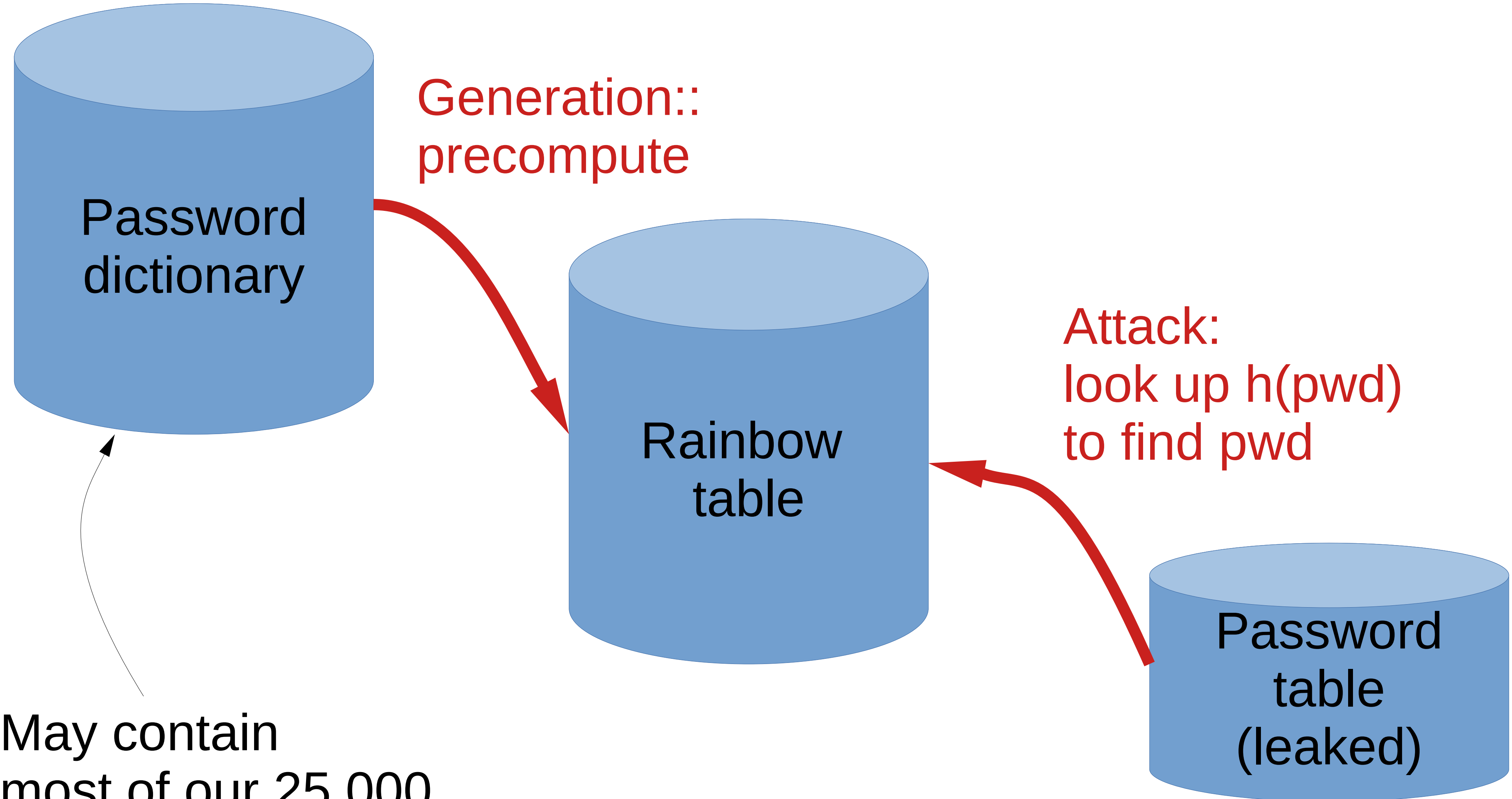
Type: offline
Target: any user account
Method: intelligent password guessing using a rainbow table

Implementation (pseudocode)

for each username u of the total of 50,000 usernames do {
  fetch the user's hashed password h in the leaked password table
  search the rainbow table to see of h is the hash value of a known password
}

A rainbow table may be extended
so as to include hashes generated
by a set of (say, 10) different, commonly used hash functions

Rainbow table attack



May contain
most of our 25,000
weak passwords
(pasta, pizza, admindnc)

Rainbow table attack: how likely is success?

It is extremely fast to do a look-up
of a hash value in a rainbow table.

If the password dictionary that the rainbow table is based on contains the password,
then the password hash, and so the password,
will be found immediately

- you can look-up 50,000 password hashes in a matter of seconds

Because rainbox table attacks are extremely dangerous,
passwords must be salted before they are hashed.

Rainbow table attack: how likely is success? (cont.)

Exercises 1 and 2
(in my paper about password-based user authentication)

In the exercises, you define a couple of passwords,
hash them, and then do a look-up in a rainbow table.

Does it appear as if the rainbow table
contains (hashes of) many passwords?

Also please discuss:

Consider two salts, say ABCD and AAAA.
Hash salt + admindnc.
How would you describe the two hash values?
In particular, are they similar to some extent?

Summary

Attack#	Attack name	Type	Target	Defenses	Conclusion
Attack #1	Passwod dictionary attack	Online	Some account	Password minimal length, “ten guesses, you’re out”	A real risk
Attack #2	Rainbow table attack	Offline	Some account	Salting	Salting prevents attack
Attack #3					

Generation:
precompute one table
for each salt !!

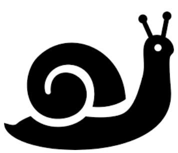
Passwords - today's plan

Introducing "PasswordbasedAuthenticator"

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted



Defense #4: The hash function must be iterative

Discussion

When passwords are hashed and salted: brute forcing an account (attack #3)

Now what can the attacker do?
- password dictionary a. (online, any account) has limited success only
- rainbow table a. (offline, any account) prevented by salting

The main, remaining threat is an offline attack
- password table has been leaked
- a resourceful attacker targets *a specific user*

```
Offline attack - brute-forcing hashed + salted passwords
repeat (say) 100,000,000 times {
   select a possible password p from very large password dictionary
   read the salt s from the password record
   compute h = sha256(s + p)
   compare h with the hash value read from the pwd record
}
```

Remaining attack
brute forcing an account (#3)

```
passwords=# select * from password;
 username |       salt       |                         hashed_password
----------+------------------+------------------------------------------------------------------
 admin    | ED5D50781D89EF20 | 4DC128AB9402D6DAE7F0EE718A8F37E8D6410722BBAF7EE5AABA6A6029B4FFF4
(1 row)
```

Resource-demanding attack
repeat (say) 100,000,000 times {
  select a possible password p from **very large password dictionary**
  read the salt s from the password record
  compute h = sha256(p + s)
  compare h with the hash value read from the pwd record
}

Is the resource-demanding attack (#3) a threat?

Unfortunately - yes!

- it is slower than a rainbow table attack (#2)
- because a rainbox table takes advantage of pre-computation
- but it is still a threat
- #3 also more dangerous, because it targets a particular account

Question 5. Implement iterative hashing. Hint: In the C# source file Hashing.cs, in the definition of method hashSHA256(), you may modify the second parameter in the call to function iteratedSha256(). What number of iterations appears to be reasonable on your computer?

Iterative hashing

step 1: compute h_1 = sha256(password + salt)

step i+1: compute h_{i+1} = sha256(h_i)

What is a “reasonably” slow hash function?

Slow - but not so slow that the legitimate user is annoyed

In your answer to Question 5, you should motivate your choice of iteration number.

Question 5 (cont.)

In your answer to question 5, your program is hashing iteratively,
as in

sha256(.. sha256(sha256(salt + password)) ..)

In a real-world application, you should use
a standard for iterative hashing,
such as “PBKDF2”

- = Password Based Key Derivation Function 2
- developed by the company RSA
- eliminates certain risks of implementing iterative hashing by hand

In C#, you can use this implementation of the standard:

- Microsoft.AspNetCore.Cryptography.KeyDerivation.Pbkdf2()
- (but you need not in the assignment)

Summary

Attack#	Attack name	Type	Target	Defenses	Conclusion
Attack #1	Passwod dictionary attack	Online	Some account	Password minimal length, “ten guesses, you’re out”	A real risk
Attack #2	Rainbow table attack	Offline	Some account	Salting	Salting prevents attack
Attack #3	Brute-forcing an account	Offline	A parti-cular account	Iterative hashing	A huge risk to weak and minimal passwords

Weak passwords (may be rejected automatically)	Minimal passwords	Sound / stronger passwords
1234 admindnc (contains username)	Only the required password length, no special characters	10+ characters, special characters

Passwords - today's plan

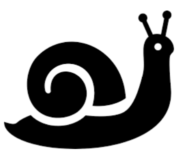
Introducing "PasswordbasedAuthenticator"

Defense #1: Passwords must have some minimal length

Defense #2: Passwords must be hashed

Defense #3: Passwords must be salted

Defense #4: The hash function must be iterative



Discussion

Memory-intensive hash functions

The iterated hash function used in defense #4
- is CPU-intensive
- forces attacker do use many parallel CPUs

However, cheap and fast hardware has emerged
for hash computation
- chips dedicated to computing, say, sha256 hashes
- used also in BitCoin mining

Percival (2009) estimated that using such specialized hardware,
an attacker can guess a 40-character passphrase password
- password has entropy 2^{56}
- with hardware costing $200,000
- in one year
- attacks is “sha256-based, iterated hash” as in defense #4
  (86,000 iterations)

The password hashing competition

A public effort study of new algorithms for *password hashing*

- 2013-2015
- algorithms also applicable to *key generation*

Algorithms evaluated -

- argon2 (the ‘competition winner’)
- scrypt (designed by Percival)
- ..

Main idea underlying the algorithms

- a hash function must be memory intensive
- not merely CPU-intensive (as sha256 and ..)

Defense #5:

- use a memory-intensive hash function
- available for C#
- but in a form that is somewhat complex to use

Assignment 5
due today 23.55

Question 4. Implement a check that the password provided by a new user at registration contains more than eight characters. Also check that the password does not contain the username. Obviously, the password ‘admindnc’ for username ‘admin’ will fail this test. Checks should be done in the definition of method passwordIsOK() in Authenticator.cs.

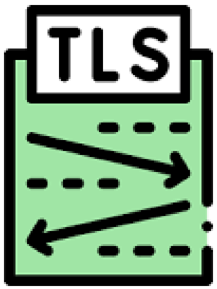
Question 5. Implement iterative hashing. Hint: In the C# source file Hashing.cs, in the definition of method hashSHA256(), you may modify the second parameter in the call to function iteratedSha256(). What number of iterations appears to be reasonable on your computer?

Question 6. In Authenticator.cs, the method sqlSetUserRecord() defines a string that is a SQL command. The SQL command is then used in method register(). Is the method vulnerable to SQL injection?

If you believe the method is vulnerable (Question 6), a good solution should include a screenshot that documents an attack.

Final security course day: Nov 12th

3. Secure sockets (TSL) (revisited)
“Forward secrecy”
(same as “Perfect forward secrecy”)



How to frequently change session key

- session key = the key for symmetric encryption

Mandatory literature:

Mark Stamp. Information Security 2/E.
Chapter 4.3.1. Textbook RSA example (97-98)
Chapter 4.4. Diffie-Helmann (100-102)
Chapter 9.3.4. Perfect Forward Secrecy (327-329)