

Assignment 4

Course code: IKT222
Semester: Autumn 2025
Number of pages: 28
Name: Bjørn Sødal

PROJECT OUTLINE

The full code repository and commit history which this report is based upon may be found in my public repository: [Repository](#)

CONTENTS

Project Outline	2
Identifying web weaknesses	3
0.1. HTTP unencrypted packets	3
0.2. Authentication bypass through SQL injection	5
0.3. Database patient record extraction through SQL injection	6
0.4. Broken access controls	7
Identifying code and database weaknesses	8
0.5. Code	8
0.5.1. Trusting user input	8
0.5.2. No proper authentication or authorization methods	9
0.6. Database	10
0.6.1. Unencrypted passwords / patient information	10
Solving vulnerabilities	11
0.7. Web	11
0.7.1. SQL model mapping	11
0.8. Code & Database	14
0.8.1. Hashing and salting passwords	14
0.8.2. Encrypting and decrypting sensitive patient information	17
Appendix A.	24
References	28

IDENTIFYING WEB WEAKNESSES

0.1. HTTP unencrypted packets. Given that this is supposedly a patient information retrieval application it should only be natural that the internet traffic web packets are encrypted through HTTPS, alas currently the application only runs in HTTP, which unless the application itself starts defining and encrypting these packets then everyone listening on the same network will be able to see the information being sent over the network.



FIGURE 1. HTTP protocol

As an example I listened on my internet traffic using Wireshark as a secondary computer accesses the running application. In the form sent one can clearly read the form contents being submitted to the application (scale PDF reader 200% for better view):

4551	280.281603	10.0.0.100	10.0.0.97	HTTP	703 POST / HTTP/1.1 (application/x-www-form-urlencoded)
4554	280.288399	10.0.0.97	10.0.0.100	HTTP	61 HTTP/1.1 200 OK
4556	280.333828	10.0.0.100	10.0.0.97	HTTP	440 GET /favicon.ico HTTP/1.1
4559	280.336711	10.0.0.97	10.0.0.100	HTTP	61 HTTP/1.1 200 OK

> Frame 4551: Packet, 703 bytes on wire (5624 bits), 703 bytes captured (5624 bits) on interface \Device\NPF_{A2D2E8D3-4C5C-4F98-A0AF-200000000000} (0.0.0.0)	0000	77 2d 66 6f 72 6d 2d 75	72 6c 65 6e 63 6f 64 65	w-form-urlencoded
> Ethernet II, Src: f2:a7:31:0c:31:1c (f2:a7:31:0c:31:1c), Dst: FourSeasGloob_34:12:9a (20:0d:b0:34:12:9a)	0000	64 0d 0e 55 70 67 72 61	64 65 2d 49 6e 73 65 63	d-Upgra de-Insec
> Internet Protocol Version 4, Src: 10.0.0.100, Dst: 10.0.0.97	0100	75 72 65 2d 52 65 71 75	65 73 74 73 3a 20 31 0d	ure-Requests: 1-
> Transmission Control Protocol, Src Port: 51450, Dst Port: 8080, Seq: 830, Ack: 4469, Len: 649	0110	0a 55 73 65 72 2d 41 67	65 6e 74 3a 20 4d 6f 7a	-User-Agent: Moz
> Hypertext Transfer Protocol	0120	69 6c 6c 61 2f 35 2e 30	20 28 57 69 6e 6a 6f 77	illa/5.0 (Window
> HTML Form URL Encoded: application/x-www-form-urlencoded	0130	73 20 4e 54 20 31 30 2e	30 3b 20 57 69 6e 6a 6f 77	s NT 10.0; Win64
> Form item: "username" = "admin"	0140	3b 20 78 36 34 20 41 70	70 70 6c 65 57 65 62 4b	; x64) A ppleiebk
> Form item: "password" = "password"	0150	69 74 2f 35 33 37 2e 33	36 20 2b 4b 48 54 4d 4c	it/537.3 e (KHTML
> Form item: "surname" = "Brown"	0160	2c 20 6c 69 6b 65 20 47	65 63 6b 6f 29 20 43 68	, like G ecko) Ch
	0170	72 6f 6d 65 2f 31 34 31	2e 30 2e 30 2e 30 20 53	rome/141.0.0.0 S
	0180	61 66 61 72 69 2f 35 33	37 2e 33 36 20 45 64 67	afari/53.7.36 Edg
	0190	2f 31 34 31 2e 30 2e 30	2e 30 0d 0a 71 61 63 62	/141.0.0.0-PC
	01a0	70 74 3a 20 74 65 78 74	2f 68 74 6d 6c 2c 61 70	pt: text/html,ap
	01b0	70 6c 69 63 61 74 69 6f	6e 2f 78 68 74 6d 6c 2b	application/xhtml
	01c0	70 6d 6c 2c 61 70 70 6c	69 63 61 74 69 6f 6e 2f	xml,applicat
	01d0	70 6d 6c 3b 71 34 30 2e	39 2c 69 6d 61 67 65 2f	xml;img, s,image/
	01e0	61 76 69 66 2c 69 6d 61	67 65 2f 77 65 62 70 2c	avif,ima ge/webp,
	01f0	69 6d 61 67 65 2f 61 70	6e 67 2c 2a 2f 2a 3b 71	image/ap ng,*/;c
	0200	3d 30 2e 30 2c 61 70 70	6c 69 63 61 74 69 6f 6e	o,8,applicat
	0210	2f 72 69 61 6e 65 64 2d	65 70 63 6d 61 6e 67 65	/signed-exchang
	0220	3b 76 34 62 33 3b 71 3d	30 2e 37 0d 0e 52 65 6e	ivb3;= 0.7.7Ref
	0230	65 72 65 72 3a 20 68 74	74 70 3a 2f 2f 31 30 2e	eren: ht tp://10.
	0240	30 2e 30 2e 39 37 3a 38	38 38 30 2f 0d 0a 41 63	0.0.97:8 080/..Ac
	0250	63 65 70 74 2d 45 6e 63	6f 64 69 6e 67 3a 20 67	cept-Enc odin: g
	0260	7a 69 70 2c 20 64 65 66	6c 61 74 65 0d 0a 41 63	zip, def late-Ac
	0270	63 65 70 74 2d 4c 61 6e	67 75 61 67 65 3a 20 65	cept-Lan guage: e
	0280	6e 2d 55 53 2c 65 6e 3b	71 3d 30 2e 39 0d 0a 0d	n-US,en; q=0.9...
	0290	0a 75 73 65 72 6e 61 6d	65 3d 61 64 61 69 6e 26	-usernam e=admin
	02a0	70 61 73 73 77 6f 72 64	3d 70 61 73 73 77 6f 72	password=passwor
	02b0	64 26 73 75 72 6e 61 6d	65 3d 42 72 6f 77 6e	d&surnam e=Brown

FIGURE 2. Form contents being submitted readable as plain text in Wireshark

In this specific conversation we can also plainly view the HTML sent from the application within the HTTP response to this very post information:

4554	280.288399	10.0.0.97	10.0.0.100	HTTP	61 HTTP/1.1 200 OK
4556	280.333828	10.0.0.100	10.0.0.97	HTTP	440 GET /favicon.ico HTTP/1.1
4559	280.336711	10.0.0.97	10.0.0.100	HTTP	61 HTTP/1.1 200 OK

> Internet Protocol Version 4, Src: 10.0.0.97, Dst: 10.0.0.100	0070	29 0d 0a 0d 0a 30 46 33	0d 0a 3c 21 6d 6f 63 7deF3 2 (docu
> Transmission Control Protocol, Src Port: 8080, Dst Port: 51450, Seq: 6370, Ack: 1479, Len: 7	0080	79 30 63 20 57 72 6d 6c	7c 3c 31 69 74 6d 6c 2b	type html><html
> [3 Reassembled TCP Segments (1908 bytes): #4552(1460), #4553(441), #4554(7)]	0090	6c 61 6e 67 3d 22 65 6e	22 3e 0a 20 20 3c 68 65	lang="en"><he
> Hypertext Transfer Protocol, has 2 chunks (including last chunk)	00a0	61 64 3e 0a 20 20 20 3c	6d 65 74 61 20 63 68	ad><meta ch
> HTTP/1.1 200 OK\r\n	00b0	61 72 73 65 74 5d 27 75	74 66 2d 38 22 3e 0a 20	arset="u tf-8">
Response Version: HTTP/1.1	00c0	20 20 20 3c 6d 65 74 61	20 6e 61 6d 65 3d 22 70	<meta name="v
Status Code: 200	00d0	69 65 77 70 6f 72 74	22 20 63 6f 6e 74 65 6e	iewport" cont
[Status Code Description: OK]	00e0	3d 22 77 69 64 74 68 3d	64 65 76 69 63 65 2d 77	="width" device-w
Response Phrase: OK	00f0	69 64 74 68 2c 20 59 6e	69 74 69 61 6c 2d 73 65	idth, in stitil:sc
Date: Sun, 26 Oct 2025 09:17:55 GMT\r\n	0100	61 6c 65 3d 11 2c 20 73	68 72 60 6e 4b 2d 74 6f	alical, s link:to
Transfer-Encoding: chunked\r\n	0110	2d 66 69 74 3d 6e 6f 22	3e 0a 20 20 20 3c 6c	-fit="no"><ll
Server: Jetty(9.4.24.v20191120)\r\n	0120	69 6e 6b 20 72 65 6c 3d	22 73 74 79 6c 65 73 68	link rel="stylesh
\r\n	0130	65 65 74 22 20 68 72 65	68 3d 22 68 74 74 70 73	et" hrc "<-http
[Request in frame: 4551]	0140	3a 2f 2f 6d 61 78 63 64	6e 2e 62 6f 6f 74 73 74	h/maxcdn n.bootst
[Time since request: 6.796000 milliseconds]	0150	72 61 70 63 64 6e 2e 63	6f 6d 2f 62 6f 6f 74 73	rapcd,c om/boots
[Request URI: /]	0160	74 72 61 70 2f 34 2e 30	2e 30 2f 63 73 2f 62	trap/4.0.0/css/b
[Full request URI: http://10.0.0.97:8080/]	0170	6f 6f 74 73 74 72 61 70	2e 6d 69 6e 2e 63 73 73	ootstrap min:sc
> HTTP chunked response	0180	22 20 69 6e 74 65 67 72	69 74 79 3d 22 73 68 61	> Integr ity><sha
> Data chunk (1779 octets)	0190	33 38 34 2d 47 6e 35 33	38 34 78 71 51 11 61 6f	384-Gn53 84xQ1a0
Chunk size: 1779 octets	01a0	57 58 41 2b 30 35 38 52	58 50 76 50 67 36 66 79	MAA+05SR XPpGsfy
Chunk data [7]: 3c21646f63747970652068746d6c3e0a3c68746d6c206c616e673d22656e223e0a20203c686561643e0a2020203c6d6574612063	01b0	44 49 57 76 54 4e 65 30	45 32 36 33 58 6d 46 63	ADW7000 E2G30x4
Chunk boundary: 0d0a	01c0	4a 6c 53 41 77 69 47 67	46 41 57 2f 64 41 69 53	l1Sawldg FAW/dA1S
> End of chunked encoding	01d0	36 4a 58 6d 22 20 63 72	6f 73 73 6f 72 69 67 69	61Xa+ cr ossorigi
Chunk size: 0 octets	01e0	6e 3d 22 61 6e 6f 6e 79	6d 6f 75 73 22 3e 0a 20	="anony mous">
\r\n	01f0	20 20 20 3c 74 69 74 6c	65 3a 50 61 74 69 65 64	<titl e><Patien
File Data: 1779 bytes	0200	74 20 52 65 63 6f 72 64	73 20 53 79 73 74 65 6d	Record s System
	0210	3a 20 50 61 74 69 65 6e	74 20 44 65 74 61 69 6c	Patien t Detail
	0220	73 3c 2f 74 69 74 6c 65	3e 0a 20 20 3c 2f 68 65	sc/<title>< /he
	0230	61 64 3e 0a 20 20 3c 6f	6f 64 79 3e 0a 20 20 3c	ad><body>

FIGURE 3. HTML page in response to form submission

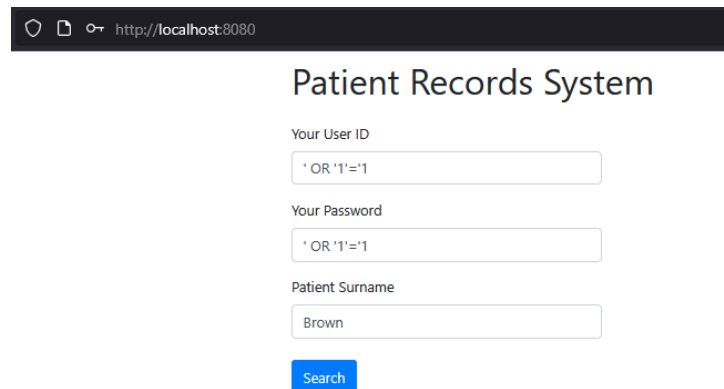
Which gives the following information:

```
1 HTTP/1.1 200 OK
2 Date: Sun, 26 Oct 2025 09:17:55 GMT
3 Transfer-Encoding: chunked
4 Server: Jetty(9.4.24.v20191120)
5
6 6F3
7 <!doctype html>
8 <html lang="en">
9   <head>
10     <meta charset="utf-8">
11     <meta name="viewport" content="width=device-width, initial-scale=1, shrink-
12       ↳ to-fit=no">
13     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap
14       ↳ /4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058
15       ↳ RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous
16       ↳ ">
17     <title>Patient Records System: Patient Details</title>
18   </head>
19   <body>
20     <div class="container">
21       <h1 class="mt-2 mb-4">Patient Records System</h1>
22       <div class="panel panel-success">
23         <div class="panel-heading">Patient Details</div>
24         <table class="table table-bordered">
25           <tr>
26             <th>Surname</th>
27             <th>Forename</th>
28             <th>Date of Birth</th>
29             <th>GP Identifier</th>
30             <th>Treated For</th>
31           </tr>
32           <tr>
33             <td>Brown</td>
34             <td>James</td>
35             <td>2016-04-10</td>
36             <td>1</td>
37             <td>Asthma</td>
38           </tr>
39         </table>
40       </div>
41       <p><a class="mt-2 btn btn-primary" role="button" href="/">Home</a></p>
42     </div>
43     <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="
44       ↳ sha384-KJ3o2DKtIkVYIK3UENzmM7KChRr/rE9/Qpg6aAZGJwFDMVNA/
45       ↳ GpGFF93hXpG5KkN" crossorigin="anonymous"></script>
46     <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/
47       ↳ popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/
48       ↳ ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>
49     <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.
50       ↳ min.js" integrity="sha384-JZR6Spejh4U02d8j0t6vLEHfe/
51       ↳ JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmY1" crossorigin="anonymous"></
52       ↳ script>
53   </body>
54 </html>
55
56 0
```

LISTING 1. Raw HTML from captured packet

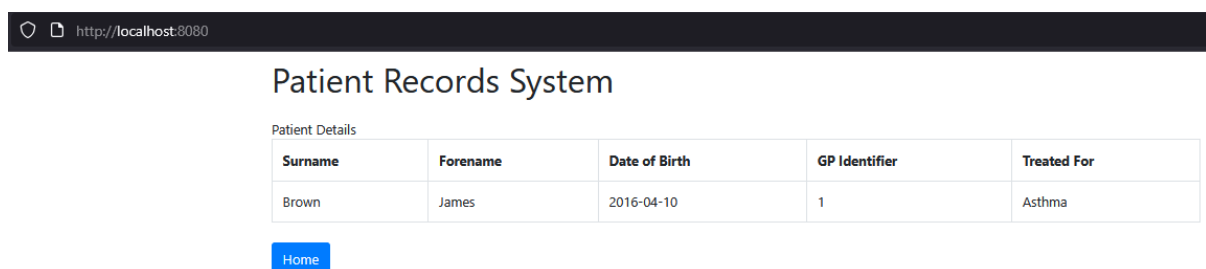
Now any actor listening on this network not only knows of a valid user account (**admin**) and its login credentials (**password**), but also just received potentially confidential patient information (**Brown**) served directly to them by simply listening in on internet traffic moving on the network. Naturally it stands to reason that in a development environment it's easier to test through HTTP, but if this application was an actively running application in production it would at minimum require a TLS/SSL certificate to enable HTTPS.

0.2. Authentication bypass through SQL injection. The web interface login form is vulnerable to SQL injections for user accounts:



The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080'. The page title is 'Patient Records System'. Below the title, there are three input fields and a 'Search' button. The first field is labeled 'Your User ID' and contains the payload ' ' OR '1'='1'. The second field is labeled 'Your Password' and also contains the payload ' ' OR '1'='1'. The third field is labeled 'Patient Surname' and contains the value 'Brown'. The 'Search' button is a blue rectangle with white text.

FIGURE 4. SQL injection into user account



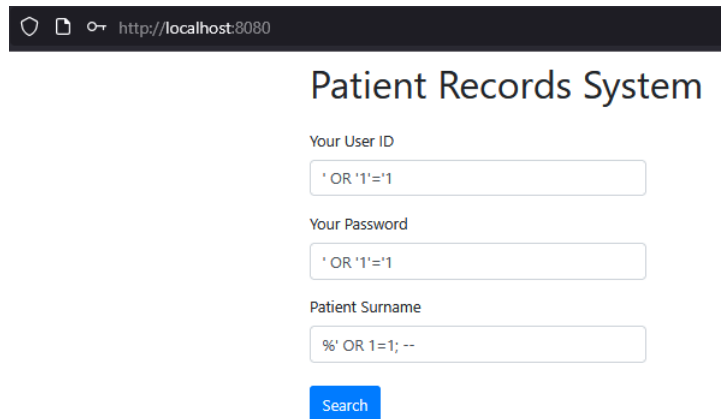
The screenshot shows the same web browser window, but now the page displays a table of patient details. The table has five columns: Surname, Forename, Date of Birth, GP Identifier, and Treated For. The first row of data shows 'Brown' as the surname, 'James' as the forename, '2016-04-10' as the date of birth, '1' as the GP identifier, and 'Asthma' as the condition treated. Below the table is a blue 'Home' button.

Surname	Forename	Date of Birth	GP Identifier	Treated For
Brown	James	2016-04-10	1	Asthma

FIGURE 5. User SQL injection result

This will allow anyone to gain access to any known patient surname record by simply circumventing the login requirement of the site. However a malicious actor may additionally circumvent having to know any surname at all as well by simply bypassing the string search of a patient name. This vulnerability was exposed by simply entering a series of standard SQL injection payloads from this GitHub page into the form fields: [Payload list](#) [1].

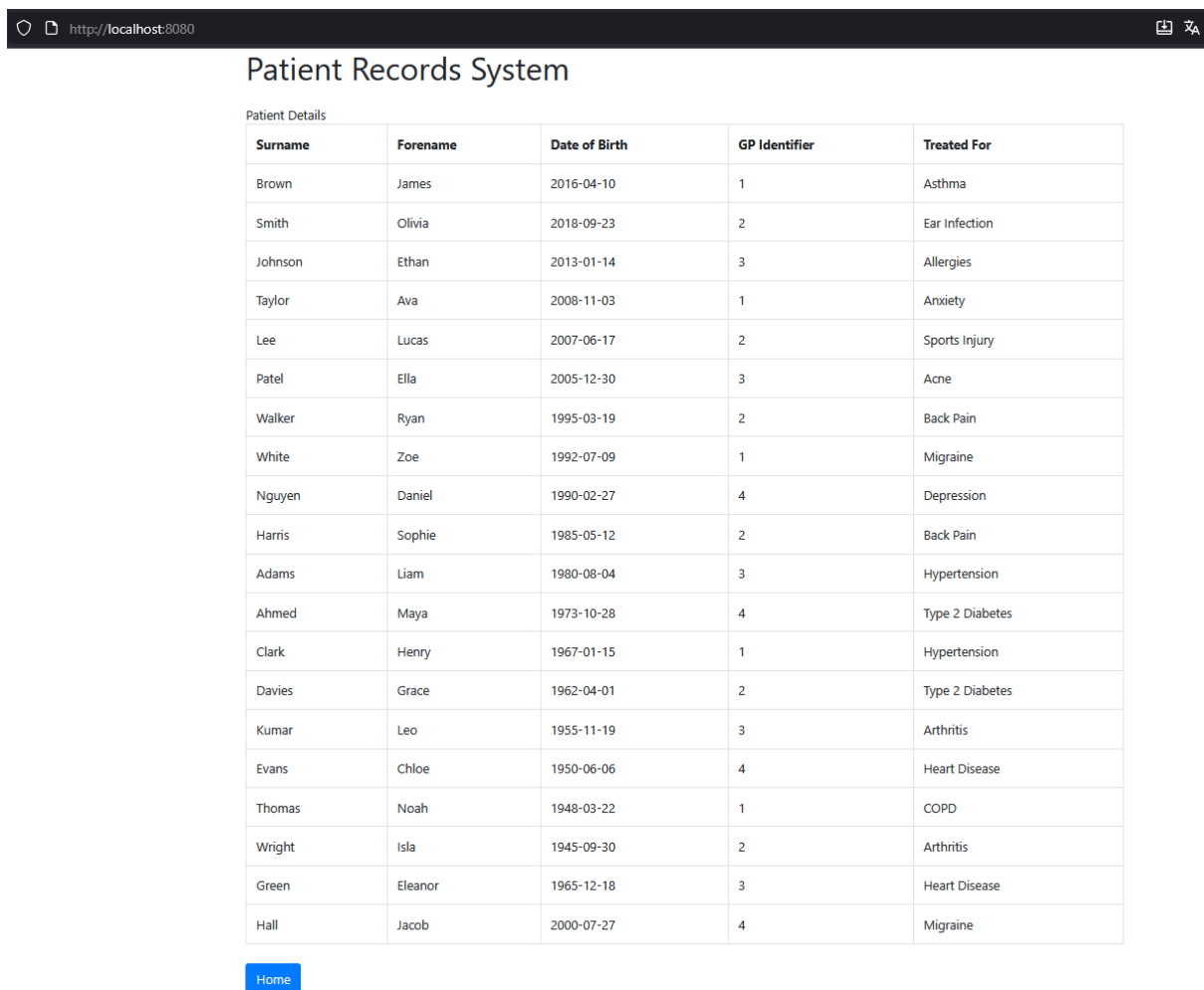
0.3. **Database patient record extraction through SQL injection.** The entire login form is actually subject to SQL injections as it appears that user input is inherently trusted within the application logic and is used as-is, letting actors extract the entire patient records table from the database.



The screenshot shows a web browser at `http://localhost:8080` displaying the "Patient Records System" login page. The page has three input fields and a "Search" button. The "Your User ID" field contains the payload `' OR '1'='1`. The "Your Password" field contains the payload `' OR '1'='1`. The "Patient Surname" field contains the payload `%' OR 1=1; --`.

Search

FIGURE 6. SQL injection to list all patient records in database



The screenshot shows the "Patient Records System" displaying a table of patient details. The table has five columns: Surname, Forename, Date of Birth, GP Identifier, and Treated For. The table contains 20 rows of patient data. Below the table is a "Home" button.

Surname	Forename	Date of Birth	GP Identifier	Treated For
Brown	James	2016-04-10	1	Asthma
Smith	Olivia	2018-09-23	2	Ear Infection
Johnson	Ethan	2013-01-14	3	Allergies
Taylor	Ava	2008-11-03	1	Anxiety
Lee	Lucas	2007-06-17	2	Sports Injury
Patel	Ella	2005-12-30	3	Acne
Walker	Ryan	1995-03-19	2	Back Pain
White	Zoe	1992-07-09	1	Migraine
Nguyen	Daniel	1990-02-27	4	Depression
Harris	Sophie	1985-05-12	2	Back Pain
Adams	Liam	1980-08-04	3	Hypertension
Ahmed	Maya	1973-10-28	4	Type 2 Diabetes
Clark	Henry	1967-01-15	1	Hypertension
Davies	Grace	1962-04-01	2	Type 2 Diabetes
Kumar	Leo	1955-11-19	3	Arthritis
Evans	Chloe	1950-06-06	4	Heart Disease
Thomas	Noah	1948-03-22	1	COPD
Wright	Isla	1945-09-30	2	Arthritis
Green	Eleanor	1965-12-18	3	Heart Disease
Hall	Jacob	2000-07-27	4	Migraine

Home

FIGURE 7. SQL injection result

This weakness where an unauthenticated user may display the entire patient records table also exposes another vulnerability issue called:

0.4. **Broken access controls.** Because users are never authenticated against some arbitrary clearance level, then any valid login attempt, such as when we inject the statement `' OR '1`, simply renders out all the patient information (see fig. 7), despite the login ID not matching the general practitioner ID in the patient table.

Patient Records System

Your User ID

Your Password

Patient Surname

FIGURE 8. Attempting to list out all patient records with non-admin user, result is same as fig. 7

This is a classic example of broken access controls as unauthenticated users should never have the privilege levels required to list out patient records not associated with themselves, or rather at all in the case of authentication bypass.

0.5. Code.

0.5.1. *Trusting user input.* When executing a search in the database for account verification the search query makes use of raw and unsanitized user input, letting users inject SQL statements into their input form field:

```

1  // Direct SQL queries (vulnerable to SQL injection attacks)
2  private static final String CONNECTION_URL = "jdbc:sqlite:db.sqlite3";
3  private static final String AUTH_QUERY = "select * from user where username
    ↳ = '%s' and password='%s'";
4  private static final String SEARCH_QUERY = "select * from patient where
    ↳ surname like '%s'";
5
6  /* ... lots of code ... */
7
8  @Override
9  protected void doPost(HttpServletRequest request, HttpServletResponse
    ↳ response)
10     throws ServletException, IOException {
11     // Get form parameters
12     String username = request.getParameter("username");
13     String password = request.getParameter("password");
14     String surname = request.getParameter("surname");
15
16     try {
17         if (authenticated(username, password)) {
18             // Get search results and merge with template
19             Map<String, Object> model = new HashMap<>();
20             model.put("records", searchResults(surname));
21             Template template = fm.getTemplate("details.html");
22             template.process(model, response.getWriter());
23         } else {
24             Template template = fm.getTemplate("invalid.html");
25             template.process(null, response.getWriter());
26         }
27         response.setContentType("text/html");
28         response.setStatus(HttpServletResponse.SC_OK);
29     } catch (Exception error) {
30         response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
31     }
32 }
33
34 private boolean authenticated(String username, String password) throws
    ↳ SQLException {
35     String query = String.format(AUTH_QUERY, username, password); // Directly
    ↳ inputting form input, i.e raw SQL string input
36     try (Statement stmt = database.createStatement()) {
37         ResultSet results = stmt.executeQuery(query);
38         return results.next();
39     }
40 }
41
42 private List<Record> searchResults(String surname) throws SQLException {
43     List<Record> records = new ArrayList<>();
44     String query = String.format(SEARCH_QUERY, surname); // Directly inputting
    ↳ surname given in form
45     try (Statement stmt = database.createStatement()) {
46         ResultSet results = stmt.executeQuery(query);
47         while (results.next()) {
48             Record rec = new Record();
49             rec.setSurname(results.getString(2));

```



```

50         rec.setForename(results.getString(3));
51         rec.setAddress(results.getString(4));
52         rec.setDateOfBirth(results.getString(5));
53         rec.setDoctorId(results.getString(6));
54         rec.setDiagnosis(results.getString(7));
55         records.add(rec);
56     }
57 }
58 return records;
59 }

```

LISTING 2. Faulty application logic

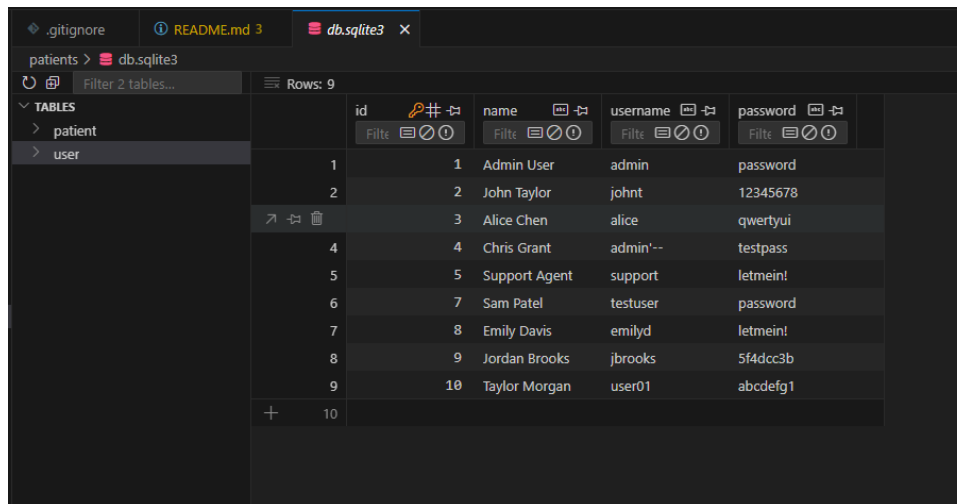
`String.format()` with the current SQL queries directly allow for SQL injections to take place, as `username`, `password` and `surname` variables are handled directly in-code as strings and not through an ORM or other sanitization means.

0.5.2. *No proper authentication or authorization methods.* Within the current application there is no session management or verification methods in place, as well as no rate-limiting on password attempts, meaning bad actors may develop a script to continuously attempt to access accounts without any form of restriction in so called brute-force attacks.

A session manager could be implemented to verify current users as well as authorize access to pages. Without a session manager every action on the application will have to require some form of login attempt, and as displayed earlier this verification method is severely flawed with its injection vulnerability.

0.6. Database.

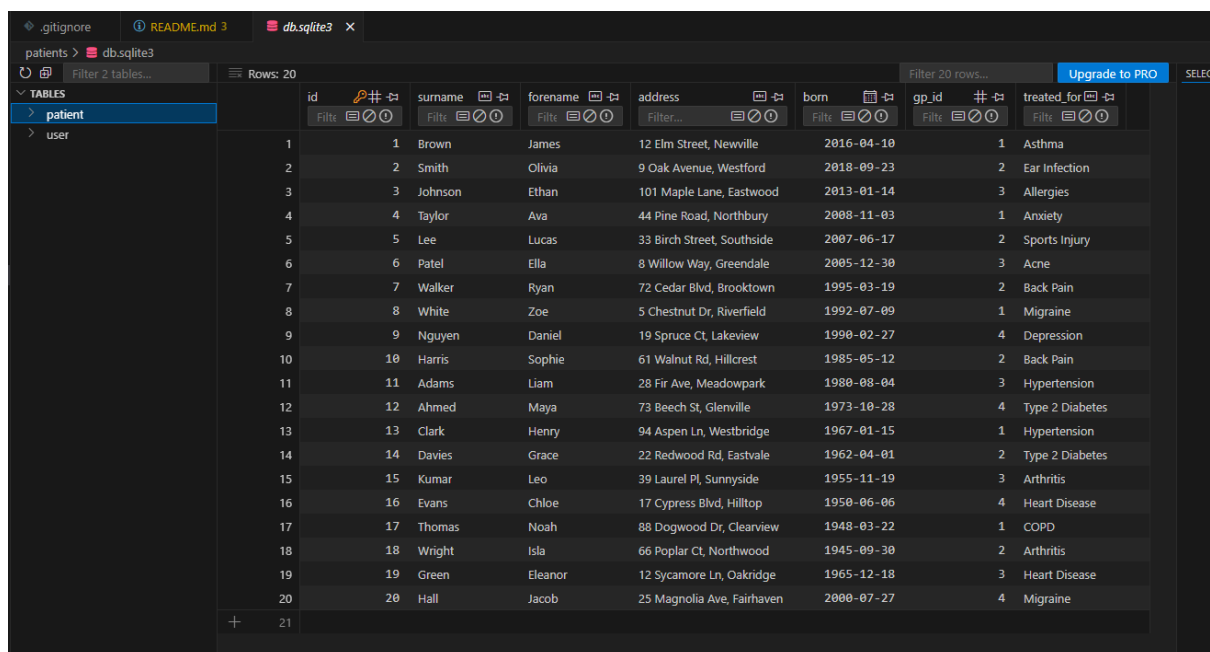
0.6.1. *Unencrypted passwords / patient information.* Within the database all the information is available as plain text, meaning in a potential breach any malicious actor would have full access to patient records as well as user account access through plain text passwords and usernames.



The screenshot shows a database viewer interface with a sidebar on the left containing a tree view with 'patients' and 'db.sqlite3'. The main area displays a table with 9 rows. The table has columns: id, name, username, and password. The data is as follows:

id	name	username	password
1	Admin User	admin	password
2	John Taylor	johnt	12345678
3	Alice Chen	alice	qwertyui
4	Chris Grant	admin'--	testpass
5	Support Agent	support	letmein!
6	Sam Patel	testuser	password
7	Emily Davis	emilyd	letmein!
8	Jordan Brooks	jbrooks	5f4dcc3b
9	Taylor Morgan	user01	abcdefg1

FIGURE 9. Plain text user information and account credentials



The screenshot shows a database viewer interface with a sidebar on the left containing a tree view with 'patients' and 'db.sqlite3'. The main area displays a table with 20 rows. The table has columns: id, surname, forename, address, born, gp_id, and treated_for. The data is as follows:

id	surname	forename	address	born	gp_id	treated_for
1	Brown	James	12 Elm Street, Newville	2016-04-10	1	Asthma
2	Smith	Olivia	9 Oak Avenue, Westford	2018-09-23	2	Ear Infection
3	Johnson	Ethan	101 Maple Lane, Eastwood	2013-01-14	3	Allergies
4	Taylor	Ava	44 Pine Road, Northbury	2008-11-03	1	Anxiety
5	Lee	Lucas	33 Birch Street, Southside	2007-06-17	2	Sports Injury
6	Patel	Ella	8 Willow Way, Greendale	2005-12-30	3	Acne
7	Walker	Ryan	72 Cedar Blvd, Brooktown	1995-03-19	2	Back Pain
8	White	Zoe	5 Chestnut Dr, Riverfield	1992-07-09	1	Migraine
9	Nguyen	Daniel	19 Spruce Ct, Lakeview	1990-02-27	4	Depression
10	Harris	Sophie	61 Walnut Rd, Hillcrest	1985-05-12	2	Back Pain
11	Adams	Liam	28 Fir Ave, Meadowpark	1980-08-04	3	Hypertension
12	Ahmed	Maya	73 Beech St, Glenville	1973-10-28	4	Type 2 Diabetes
13	Clark	Henry	94 Aspen Ln, Westbridge	1967-01-15	1	Hypertension
14	Davies	Grace	22 Redwood Rd, Eastvale	1962-04-01	2	Type 2 Diabetes
15	Kumar	Leo	39 Laurel Pl, Sunnyside	1955-11-19	3	Arthritis
16	Evans	Chloe	17 Cypress Blvd, Hilltop	1950-06-06	4	Heart Disease
17	Thomas	Noah	88 Dogwood Dr, Clearview	1948-03-22	1	COPD
18	Wright	Isla	66 Poplar Ct, Northwood	1945-09-30	2	Arthritis
19	Green	Eleanor	12 Sycamore Ln, Oakridge	1965-12-18	3	Heart Disease
20	Hall	Jacob	25 Magnolia Ave, Fairhaven	2000-07-27	4	Migraine

FIGURE 10. Plain text patient information

To at least prevent user account access credentials from being leaked in a potential breach, at minimum the passwords should be encrypted and salted, and ideally all identifiable patient record data should be encrypted to prevent sensitive data from being exploited to prey upon vulnerable patients in the case of a breach or leakage.

0.7. Web. Perhaps the most pressing issue to solve for the web interface would be the SQL injection vulnerability (though HTTPS enforcement is also very high on the list), as this one vulnerability exposes the entire application and its contents to whoever is savvy enough to attempt injecting into it. A relatively quick solution to this problem given the MVC, Model View Controller, architecture of this application is simply passing the Java models (User and Patient) definitions through an ORM which will handle all queries as SQL mappings and sanitize untrustworthy user input for us.

0.7.1. SQL model mapping. The purpose of this implementation is to eliminate the use of direct SQL queries and to sanitize user input. To do this it's standard practice to stop treating user inputs as literal strings which get implemented straight into the SQL queries, but instead parametrize them and treat them like data instead. This is the basic concept behind an ORM, but to avoid difficult overhead and implementation of large packages such as MyBatis [2] I've elected to perform straight SQL mapping through Java's own JDK methods while sanitizing input.

Staying in line with SQL mapping we'll keep the current SQL queries but exchange its parameters from pure strings to placeholders:

```
1 private static final String AUTH_QUERY = "select * from user where username=?
    ↳ and password=?";
2 private static final String SEARCH_QUERY = "select * from patient where surname
    ↳ like ?";
```

LISTING 3. Redefining SQL queries

One may then redefine the `authenticated` method to make use of these new parameters and query execution:

```
1 private boolean authenticated(String username, String password) throws
    ↳ SQLException {
2
3     // Prepare the SQL statement with placeholders
4     try (java.sql.PreparedStatement pstmt = database.prepareStatement(
        ↳ AUTH_QUERY)) {
5
6         // Bind the user input to the placeholders
7         // Driver handles escaping and treats ' or '1=1-- as literal strings
8         pstmt.setString(1, username); // Binds 'username' to the first '?'
9         pstmt.setString(2, password); // Binds 'password' to the second '?'
10
11        // Execute defined query
12        try (ResultSet results = pstmt.executeQuery()) {
13            // Check if any row was returned
14            return results.next();
15        }
16    }
17 }
```

LISTING 4. New authenticated with parameterized and sanitized user input

Removing `String.format()` and `Statement` with a `PreparedStatement` treats the user input as data, not literal strings, ensuring input such as `' OR '1` don't become actual executable SQL parameters. This is the same approach I'll use for the `searchResults()` method:

```
1 private List<Record> searchResults(String surname) throws SQLException {
2     List<Record> records = new ArrayList<>();
3
4     // Prepare the SQL statement with a placeholder
5     try (java.sql.PreparedStatement pstmt = database.prepareStatement(
        ↳ SEARCH_QUERY)){
6
7         // Bind the user input to the placeholder, manually adding wildcards
8         // as PreparedStatement only protects user input itself
9         pstmt.setString(1, '%' + surname + "%");
10 }
```

```

11      // Execute defined query
12      try(ResultSet results = pstmt.executeQuery()){
13          while (results.next()) {
14              Record rec = new Record();
15              rec.setSurname(results.getString(2));
16              rec.setForename(results.getString(3));
17              rec.setAddress(results.getString(4));
18              rec.setDateOfBirth(results.getString(5));
19              rec.setDoctorId(results.getString(6));
20              rec.setDiagnosis(results.getString(7));
21              records.add(rec);
22          }
23      }
24  }
25  return records;
26  }

```

LISTING 5. New searchResults with parameterized and sanitized user input

With these sanitization methods in place and SQL model mapping against the database, SQL injections no longer work:

Patient Records System

Your User ID

Your Password

Patient Surname

FIGURE 11. Testing SQL injection against new parameterized queries and sanitization

Patient Records System

The login credentials you supplied are not valid.
Please **try again**.

FIGURE 12. SQL injection with new sanitization and mapping results

To fully test a bunch of payloads from the GitHub page I made a small python script to input these payloads into the form fields, and if the redirect went somewhere other than the non-valid page then the SQL injection would be a success. With proper rate limiting and session management this test script would not be possible, highlighting the necessity of proper validation and authentication methods in this application. The full output of this testing script can be found in the appendix, see listing 15:

```

1  import requests
2  from bs4 import BeautifulSoup
3  import sys
4  import time
5

```

```

6  # URL of running application
7  BASE_URL = "http://localhost:8080/"
8
9  # Failure message displayed on the invalid login page
10 FAILURE_TEXT = "The login credentials you supplied are not valid."
11
12 # Bunch of SQL injection payloads to test: https://github.com/payloadbox/sql-
    ↳ injection-payload-list
13 PAYLOADS = [...]
14
15 def normalize_payload(p):
16     return p.split("\t", 1)[0]
17
18 PAYLOADS = [normalize_payload(p) for p in PAYLOADS]
19
20 session = requests.Session()
21 session.headers.update({
22     "User-Agent": "sqli-tester/1.0",
23 })
24
25 # Bunch of function definitions to input payloads and respond to site changes,
    ↳ see project repository for full python file (injection.py)
26
27 def functions():
28     ....
29
30 def main():
31     print("Starting SQL injection payload test against", BASE_URL)
32     for p in PAYLOADS:
33         payload = p
34         print(f"Testing payload: {payload!r}")
35         resp = submit_payload(payload)
36         # If we get the failure page (or redirect to it) -> print failed
37         if is_failure_page(resp):
38             print(f"Injection {payload} failed")
39             # follow the try again link if present so next POST goes to the
                ↳ form page
40             follow_try_again_link(resp)
41             # small pause to avoid aggressive hammering
42             time.sleep(0.2)
43             continue
44             # If not a failure page, check status or heuristics indicating success
45             # (e.g., maybe the app returned a search result page / no failure text)
46         if resp is None:
47             print(f"Injection {payload} result: request error")
48             continue
49             # Heuristic: if we are still on the root page (form HTML) but without
                ↳ failure text,
50             # it could be a different response. Print a message and continue.
51         body = resp.text or ""
52         if FAILURE_TEXT not in body:
53             print(f"Injection {payload} may have succeeded or returned a non-
                ↳ failure response (inspect manually)")
54             # Stop after first potential success
55             break
56         else:
57             print(f"Injection {payload} failed")
58             time.sleep(0.2)
59
60 if __name__ == "__main__":
61     main()

```

LISTING 6. Python script to test SQL injections

0.8. Code & Database. For the code and database vulnerability the largest safety measure to take would be to secure the contents of the database entirely. Even large companies fail to completely inhibit attackers from getting access to their databases, and as such it is highly unlikely that this application will succeed either. As such, focusing on securing its contents and minimizing useful login credentials and personal information from being leaked is of utmost priority.

0.8.1. Hashing and salting passwords. Given this is a test application I've elected not to create an entire sign up / register page and form for the application, but rather focused on salting and hashing the current passwords stored in the database, as well as checking user login credentials against these hashed and salted passwords. For encryptions and salting I've made use of `BCrypt` [3]. To make use of `BCrypt` we must first import it after adding its dependency into the `build.gradle` file:

```
1 // BCrypt library for password hashing
2 import org.mindrot.jbcrypt.BCrypt;
```

LISTING 7. jBCrypt dependency import

With `BCrypt` now exposed to the application one may fetch only the stored hashed password for a given username:

```
1 // Updated AUTH_QUERY to select hashed password
2 private static final String AUTH_QUERY = "select password from user where
    ↳ username=?";
```

LISTING 8. Update AUTH_QUERY

Now all that's left is to make sure the `authenticated()` function makes use of `BCrypt.checkpw()`:

```
1 private boolean authenticated(String username, String password) throws
    ↳ SQLException {
2
3     // Prepare the SQL statement with placeholders
4     try (java.sql.PreparedStatement pstmt = database.prepareStatement(
        ↳ AUTH_QUERY)) {
5
6         // Bind the user input to the placeholders
7         // Driver handles escaping and treats ' or '!=1-- as literal strings
8         pstmt.setString(1, username); // Binds 'username' to the first '?'
9
10
11        // Execute defined query
12        try (ResultSet results = pstmt.executeQuery()) {
13            // Check if a user with that username was found
14            if (results.next()) {
15                // User found, get the stored hashed password
16                String storedHash = results.getString("password");
17
18                // Verify the provided password against the stored hash
19                // BCrypt.checkpw handles all the salt comparison logic
20                return BCrypt.checkpw(password, storedHash);
21            } else {
22                // No such user found
23                return false;
24            }
25        }
26    }
27 }
```

LISTING 9. Modified `authenticated()` to use `BCrypt` check password hash

Now the login form will make use of the hashed and salted passwords for matching. However, since these changes haven't actually changed the passwords themselves already stored in the database from plain text to hashed we need a temporary Java file `HashPasswords.java` to hash all the passwords in the database which will run only once:

```

1 package IKT222.Assignment4;
2
3 import java.sql.Connection;
4 import java.sql.DriverManager;
5 import java.sql.PreparedStatement;
6 import java.sql.ResultSet;
7 import java.sql.SQLException;
8 import java.sql.Statement;
9 import java.util.HashMap;
10 import java.util.Map;
11 import org.mindrot.jbcrypt.BCrypt;
12
13 /*
14  * Utility to hash all plaintext passwords in the 'user' table of the database.
15  * This will only be run once to hash all password entries in the user table.
16  */
17
18 public class HashPasswords {
19     private static final String CONNECTION_URL = "jdbc:sqlite:db.sqlite3";
20     private static final String SELECT_USERS_QUERY = "SELECT username, password
21     ↪ FROM user";
22     private static final String UPDATE_PASSWORD_QUERY = "UPDATE user SET
23     ↪ password = ? WHERE username = ?";
24
25     public static void main(String[] args) {
26         System.out.println("Starting password hashing utility...");
27         Connection connection = null;
28         Statement selectStatement = null;
29         PreparedStatement updateStatement = null;
30         ResultSet users = null;
31
32         // This map will store username -> plaintext password
33         Map<String, String> userPasswords = new HashMap<>();
34
35         try {
36             // Connect to the database
37             Class.forName("org.sqlite.JDBC");
38             connection = DriverManager.getConnection(CONNECTION_URL);
39             // Use transactions for batch updates, allows rollback on potential
40             ↪ errors
41             connection.setAutoCommit(false);
42
43             System.out.println("Connected to database.");
44
45             // Select all users and their current passwords
46             selectStatement = connection.createStatement();
47             users = selectStatement.executeQuery(SELECT_USERS_QUERY);
48
49             System.out.println("Fetching users...");
50             while (users.next()) {
51                 String username = users.getString("username");
52                 String plaintextPassword = users.getString("password");
53
54                 // Check if password looks like a hash already
55                 if (plaintextPassword.startsWith("$2a$")) {
56                     System.out.println("Skipping user '" + username + "':
57                     ↪ password already looks hashed.");
58                 } else {
59                     userPasswords.put(username, plaintextPassword);
60                 }
61             }
62         }
63     }
64 }

```

```

58     System.out.println("Found " + userPasswords.size() + " users with
        ↪ plaintext passwords.");
59
60     //Hash and update passwords
61     if (userPasswords.isEmpty()) {
62         System.out.println("No plaintext passwords to update.");
63         return;
64     }
65
66     updateStatement = connection.prepareStatement(UPDATE_PASSWORD_QUERY
        ↪ );
67
68     int count = 0;
69     for (Map.Entry<String, String> entry : userPasswords.entrySet()) {
70         String username = entry.getKey();
71         String plaintextPassword = entry.getValue();
72
73         // Generate a salt and hash the password (work factor 12: https
        ↪ ://github.com/jeremyh/jBCrypt)
74         String hashedPassword = BCrypt.hashpw(plaintextPassword, BCrypt
        ↪ .gensalt(12));
75
76         // Set parameters for the update query
77         updateStatement.setString(1, hashedPassword);
78         updateStatement.setString(2, username);
79
80         // Add to batch
81         updateStatement.addBatch();
82         count++;
83
84         System.out.println("Queued update for user: " + username);
85     }
86
87     // Execute the batch update
88     System.out.println("Executing batch update for " + count + " users
        ↪ ...");
89     int[] updateCounts = updateStatement.executeBatch();
90     connection.commit(); // Commit transaction
91
92     System.out.println("Successfully updated " + updateCounts.length +
        ↪ " passwords.");
93     System.out.println("Password hashing complete.");
94
95     } catch (SQLException | ClassNotFoundException e) {
96         System.err.println("An error occurred: " + e.getMessage());
97         e.printStackTrace();
98         if (connection != null) {
99             try {
100                 System.err.println("Rolling back transaction...");
101                 connection.rollback();
102             } catch (SQLException ex) {
103                 System.err.println("Error during rollback: " + ex.
                    ↪ getMessage());
104             }
105         }
106     } finally {
107         // Clean up
108         try { if (users != null) users.close(); } catch (SQLException e) {
            ↪ /* ignored */ }
109         try { if (selectStatement != null) selectStatement.close(); } catch
            ↪ (SQLException e) { /* ignored */ }

```



```

110     try { if (updateStatement != null) updateStatement.close(); } catch
111         ↳ (SQLException e) { /* ignored */ }
112     try { if (connection != null) connection.close(); } catch (
113         ↳ SQLException e) { /* ignored */ }
114     System.out.println("Database connection closed.");
115 }

```

LISTING 10. HashPasswords.java

The screenshot displays a database viewer interface for a SQLite3 database. The 'user' table is selected, showing 9 rows of data. The columns are id, name, username, and password. The password column contains hashed and salted passwords. Below the table, a terminal window shows the output of running the HashPasswords.java application.

id	name	username	password
1	Admin User	admin	\$2a\$12\$kA5dTi/4tlhmbq9duvSfROjT0YjG7lh.pNYTh89X...
2	John Taylor	johnt	\$2a\$12\$mRtPBqyoczgY7oOiFG77.PQFaoGDbqe16XuTay...
3	Alice Chen	alice	\$2a\$12\$Wg25U61k.K/kgXtk.GCU5uCB70Hwbx9qU94va...
4	Chris Grant	admin'--	\$2a\$12\$G9V.Zm.gdqTgjFfoj6buc8tZjV5741.I9npj9piOi...
5	Support Agent	support	\$2a\$12\$35J5VaYegM0Qkj4czeRhEeMCHks1BPZPBli8FSRJ...
6	Sam Patel	testuser	\$2a\$12\$T16XPDeul2rvlg0NkfDqjeuFkKSg1tmUjTAPuzCz...
7	Emily Davis	emilyd	\$2a\$12\$XLiiz4uKSAgy/9U7IPHVONRzBvrvOAaYg/Cn3bH...
8	Jordan Brooks	jbrooks	\$2a\$12\$18n/NhzZKHabPmXVFJSJlUMLpVstaTFYB1T/You...
9	Taylor Morgan	user01	\$2a\$12\$Q20VlPcxYFrXKzy5LjuX.c6OCZ3biTlsGrB/2JXw7...

```

PS C:\Users\bjorn\temp\IKT222-Assignment-4\patients> ./gradlew run
> Task :run
Starting password hashing utility...
Connected to database.
Fetching users...
Found 9 users with plaintext passwords.
Queued update for user: johnt
Queued update for user: admin'--
Queued update for user: admin'--
Queued update for user: user01
Queued update for user: jbrooks
Queued update for user: admin
Queued update for user: admin
Queued update for user: alice
Queued update for user: alice
Queued update for user: testuser
Queued update for user: testuser
Queued update for user: emilyd
Queued update for user: emilyd
Queued update for user: support
Queued update for user: support
Executing batch update for 9 users...
Successfully updated 9 passwords.
Successfully updated 9 passwords.
Password hashing complete.
Password hashing complete.
Database connection closed.

```

FIGURE 13. Hashed and salted passwords after running HashPasswords.java

0.8.2. *Encrypting and decrypting sensitive patient information.* Because this data needs to be viewed in plain text again once a general practitioner has searched them up, one cannot make use of a hashing algorithm as these are non-reversible encryptions. Instead we'll have to make use of a reversible cipher to encrypt the sensitive patient columns so that when a general practitioner requests a particular patient history the application may reverse the cipher and procure the plain text anew. Ordinarily one would encrypt the entirety of the identifiable patient information columns, but because the search query in the main application makes use of the `surname` column as a search index I've elected to only encrypt

forname, address, born and treated_for to minimize the amount of possibly identifiable information tied to a patient in the case of a breach.

This encryption will be making use of AES, Advanced Encryption Standard [4], as it is a symmetric key algorithm standardized as part of the JDK as well as a good fit for this encryption purpose. Ordinarily one would be hiding away the secret key in an environment file or similar methods, but for simplicity's sake I'll be hard coding it into the application for encrypting the relevant columns in the patient table as well as decrypting them.

For the cipher mode I've elected to make use of CBC, Cipher Block Chaining, with PKCS5Padding. To encrypt the columns I approached the process in the same manner as hashing and salting passwords, by creating a separate java file and executing it once:

```
1 package IKT222.Assignment4;
2
3 import javax.crypto.Cipher;
4 import javax.crypto.spec.SecretKeySpec;
5 import javax.crypto.spec.IvParameterSpec;
6 import java.security.SecureRandom;
7 import java.sql.Connection;
8 import java.sql.DriverManager;
9 import java.sql.PreparedStatement;
10 import java.sql.ResultSet;
11 import java.sql.SQLException;
12 import java.sql.Statement;
13 import java.util.Base64;
14 import java.nio.charset.StandardCharsets;
15 import java.util.ArrayList;
16 import java.util.List;
17
18 // Simple class to hold patient data and ID before encryption
19 class PatientRecord {
20     int id;
21     String forename;
22     String born;
23     String address;
24     String treatedfor;
25
26     public PatientRecord(int id, String forename, String born, String address,
27         ↪ String treatedfor) {
28         this.id = id;
29         this.forename = forename;
30         this.born = born;
31         this.address = address;
32         this.treatedfor = treatedfor;
33     }
34 }
35
36 public class EncryptPatients {
37     // Do not use hardcoded keys in production systems! This is just for
38     ↪ exemplification.
39     private static final String SECRET_KEY = "SupastrongKey123";
40     private static final String ALGORITHM = "AES";
41     private static final String TRANSFORMATION = "AES/CBC/PKCS5Padding";
42     private static final String CONNECTION_URL = "jdbc:sqlite:db.sqlite3";
43
44     // SQL queries
45     private static final String SELECT_PATIENTS_QUERY = "SELECT id, forename,
46         ↪ born, address, treated_for FROM patient";
47     private static final String UPDATE_PATIENT_QUERY = "UPDATE patient SET
48         ↪ forename = ?, born = ?, address = ?, treated_for = ? WHERE id = ?";
49
50     private static SecretKeySpec getKeySpec() {
```

```

47         return new SecretKeySpec(SECRET_KEY.getBytes(StandardCharsets.UTF_8),
48             ↪ ALGORITHM);
49     }
50     public static String encrypt(String plainText) throws Exception {
51         if (plainText == null || plainText.isEmpty()) {
52             return null;
53         }
54
55         Cipher cipher = Cipher.getInstance(TRANSFORMATION);
56         byte[] iv = new byte[16];
57         new SecureRandom().nextBytes(iv);
58         IvParameterSpec ivSpec = new IvParameterSpec(iv);
59
60         cipher.init(Cipher.ENCRYPT_MODE, getKeySpec(), ivSpec);
61
62         byte[] encryptedBytes = cipher.doFinal(plainText.getBytes(
63             ↪ StandardCharsets.UTF_8));
64
65         // Prepend the IV to the ciphertext
66         byte[] combined = new byte[iv.length + encryptedBytes.length];
67         System.arraycopy(iv, 0, combined, 0, iv.length);
68         System.arraycopy(encryptedBytes, 0, combined, iv.length, encryptedBytes
69             ↪ .length);
70
71         return Base64.getEncoder().encodeToString(combined);
72     }
73
74     public static void main(String[] args) {
75         System.out.println("Starting patient data encryption utility...");
76         Connection connection = null;
77         Statement selectStatement = null;
78         PreparedStatement updateStatement = null;
79         ResultSet patients = null;
80
81         // Collect patient records to encrypt
82         List<PatientRecord> patientsToEncrypt = new ArrayList<>();
83
84         try {
85             // Connect to the database (Same as HashPasswords.java)
86             Class.forName("org.sqlite.JDBC");
87             connection = DriverManager.getConnection(CONNECTION_URL);
88             // Use transactions for batch updates, allows rollback on potential
89             ↪ errors
90             connection.setAutoCommit(false);
91
92             System.out.println("Connected to database.");
93
94             // Select all patients
95             selectStatement = connection.createStatement();
96             patients = selectStatement.executeQuery(SELECT_PATIENTS_QUERY);
97
98             System.out.println("Fetching patient records...");
99             while (patients.next()) {
100                 int id = patients.getInt("id");
101                 String forename = patients.getString("forename");
102
103                 // Simple check to skip already encrypted records (encrypted
104                 ↪ data is Base64, so it should contain '=')
105                 if (forename != null && !forename.contains("=")) {
106                     patientsToEncrypt.add(new PatientRecord(
107                         id,

```

```

104         forename,
105         patients.getString("born"),
106         patients.getString("address"),
107         patients.getString("treated_for")
108     ));
109     }
110 }
111 System.out.println("Found " + patientsToEncrypt.size() + " patients
    ↳ with plaintext data to encrypt.");
112
113 if (patientsToEncrypt.isEmpty()) {
114     System.out.println("No plaintext records to update. Encryption
    ↳ utility exiting.");
115     return;
116 }
117
118 // Encrypt and prepare batch update
119 updateStatement = connection.prepareStatement(UPDATE_PATIENT_QUERY)
    ↳ ;
120 int count = 0;
121
122 for (PatientRecord record : patientsToEncrypt) {
123     String encryptedForename = encrypt(record.forename);
124     String encryptedborn = encrypt(record.born);
125     String encryptedAddress = encrypt(record.address);
126     String encryptedTreatedFor = encrypt(record.treatedfor);
127
128     updateStatement.setString(1, encryptedForename);
129     updateStatement.setString(2, encryptedborn);
130     updateStatement.setString(3, encryptedAddress);
131     updateStatement.setString(4, encryptedTreatedFor);
132     updateStatement.setInt(5, record.id);
133     updateStatement.addBatch();
134     count++;
135 }
136
137 // Execute the batch update and commit
138 System.out.println("Executing batch update for " + count + "
    ↳ records...");
139 int[] updateCounts = updateStatement.executeBatch();
140 connection.commit();
141
142 System.out.println("Successfully encrypted and updated " +
    ↳ updateCounts.length + " patient records.");
143
144 } catch (Exception e) {
145     System.err.println("An error occurred: " + e.getMessage());
146     e.printStackTrace();
147     if (connection != null) {
148         try {
149             System.err.println("Rolling back transaction...");
150             connection.rollback();
151         } catch (SQLException ex) {
152             System.err.println("Error during rollback: " + ex.
    ↳ getMessage());
153         }
154     }
155 } finally {
156     // Clean up
157     try {
158         if (patients != null) patients.close();
159     } catch (SQLException e) { /* ignored */ }

```



```

15     if (combined.length < IV_LENGTH) {
16         throw new IllegalArgumentException("Encrypted data is too short to
           ↳ contain a valid IV.");
17     }
18
19     byte[] iv = new byte[IV_LENGTH];
20     System.arraycopy(combined, 0, iv, 0, IV_LENGTH);
21     IvParameterSpec ivSpec = new IvParameterSpec(iv);
22
23     int cipherTextLength = combined.length - IV_LENGTH;
24     byte[] cipherText = new byte[cipherTextLength];
25     System.arraycopy(combined, IV_LENGTH, cipherText, 0, cipherTextLength);
26
27     Cipher cipher = Cipher.getInstance(TRANSFORMATION);
28     cipher.init(Cipher.DECRYPT_MODE, getKeySpec(), ivSpec);
29
30     byte[] decryptedBytes = cipher.doFinal(cipherText);
31     return new String(decryptedBytes, StandardCharsets.UTF_8);
32 }

```

LISTING 13. Helper function and decryption definition

```

1  // Modified to decrypt patient data before returning web page
2  private List<Record> searchResults(String surname) throws SQLException {
3      List<Record> records = new ArrayList<>();
4
5      // Prepare the SQL statement with a placeholder
6      try (PreparedStatement pstmt = database.prepareStatement(SEARCH_QUERY)) {
7
8          // Bind the user input to the placeholder, manually adding wildcards
9          // as PreparedStatement only protects user input itself
10         pstmt.setString(1, '%' + surname + "%");
11
12         // Execute defined query
13         try (ResultSet results = pstmt.executeQuery()) {
14             while (results.next()) {
15                 Record rec = new Record();
16                 String errorMsg = "DECRYPTION FAILED";
17
18                 // Plaintext data
19                 rec.setSurname(results.getString(2));
20                 rec.setDoctorId(results.getString(6));
21
22                 // Retrieve encrypted data
23                 String encryptedForename = results.getString(3);
24                 String encryptedAddress = results.getString(4);
25                 String encryptedBorn = results.getString(5);
26                 String encryptedTreatedFor = results.getString(7);
27
28                 try {
29                     // Decrypting and setting sensitive columns
30                     rec.setForename(decrypt(encryptedForename));
31                     rec.setAddress(decrypt(encryptedAddress));
32                     rec.setDateOfBirth(decrypt(encryptedBorn));
33                     rec.setDiagnosis(decrypt(encryptedTreatedFor));
34                 } catch (Exception e) {
35                     System.err.println(
36                         "Decryption failed for a record. Data might be corrupted or key
                           ↳ is wrong: " + e.getMessage());
37                     // Set fields to an error message or null if decryption fails
38                     rec.setForename(errorMsg);
39                     rec.setAddress(errorMsg);

```

```

40         rec.setDateOfBirth(errorMsg);
41         rec.setDiagnosis(errorMsg);
42     }
43     records.add(rec);
44 }
45 }
46 }
47 return records;
48 }

```

LISTING 14. Updated searchResults to decrypt encrypted columns before display

With this decryption method in order the database is now relatively secretive, and as such even if a malicious actor were to gain access to it, would retrieve minimally useful data to exploit or piece together, all while keeping the application functionality the exact same:

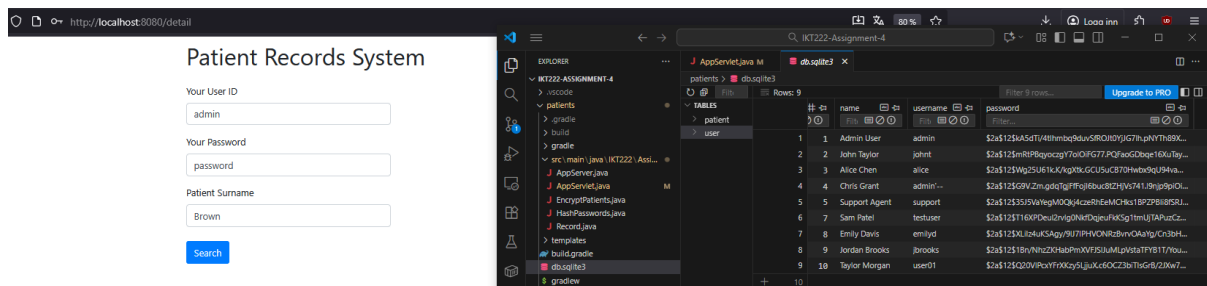


FIGURE 15. Testing application with encrypted database information (zoom 250% for better resolution)

Patient Records System

Patient Details

Surname	Forename	Date of Birth	GP Identifier	Treated For
Brown	James	2016-04-10	1	Asthma

[Home](#)

FIGURE 16. Result of test

APPENDIX A.

```
1 Starting SQL injection payload test against http://localhost:8080/
2 Testing payload: ""
3 Injection ' failed
4 Testing payload: ""
5 Injection '' failed
6 Testing payload: ''
7 Injection ' failed
8 Testing payload: ''
9 Injection '' failed
10 Testing payload: ', '
11 Injection , failed
12 Testing payload: '" '
13 Injection " failed
14 Testing payload: '" "'
15 Injection "" failed
16 Testing payload: '/'
17 Injection / failed
18 Testing payload: '// '
19 Injection // failed
20 Testing payload: '\\ '
21 Injection \ failed
22 Testing payload: '\\\\ '
23 Injection \\ failed
24 Testing payload: ';'
25 Injection ; failed
26 Testing payload: '\ ' or " '
27 Injection ' or " failed
28 Testing payload: '-- or #'
29 Injection -- or # failed
30 Testing payload: "' OR '1"
31 Injection ' OR '1 failed
32 Testing payload: "' OR 1 -- -"
33 Injection ' OR 1 -- - failed
34 Testing payload: '" OR "" = "'
35 Injection " OR "" = " failed
36 Testing payload: '" OR 1 = 1 -- -'
37 Injection " OR 1 = 1 -- - failed
38 Testing payload: "' OR '' = '"
39 Injection ' OR '' = ' failed
40 Testing payload: "'=' "
41 Injection '=' failed
42 Testing payload: "'LIKE'"
43 Injection 'LIKE' failed
44 Testing payload: "'=0--+"
45 Injection '=0--+ failed
46 Testing payload: ' OR 1=1'
47 Injection OR 1=1 failed
48 Testing payload: "' OR 'x'='x"
49 Injection ' OR 'x'='x failed
50 Testing payload: "' AND id IS NULL; --"
51 Injection ' AND id IS NULL; -- failed
52 Testing payload: "''''''''''''''''UNION SELECT '2"
53 Injection ''''''''''''''''UNION SELECT '2 failed
54 Testing payload: '- '
55 Injection - failed
56 Testing payload: ' '
57 Injection  failed
58 Testing payload: '&'
59 Injection & failed
60 Testing payload: '^'
```



```

61 Injection ^ failed
62 Testing payload: '*'
63 Injection * failed
64 Testing payload: " or '-"
65 Injection or '- failed
66 Testing payload: " or ' '"
67 Injection or ' ' failed
68 Testing payload: " or '&'"
69 Injection or '&' failed
70 Testing payload: " or '^'"
71 Injection or '^' failed
72 Testing payload: " or '*''"
73 Injection or '*' failed
74 Testing payload: '- '
75 Injection - failed
76 Testing payload: ' '
77 Injection failed
78 Testing payload: '&'
79 Injection & failed
80 Testing payload: '^ '
81 Injection ^ failed
82 Testing payload: '*'
83 Injection * failed
84 Testing payload: ' or "- '
85 Injection or "- failed
86 Testing payload: ' or " " '
87 Injection or " " failed
88 Testing payload: ' or "&" '
89 Injection or "&" failed
90 Testing payload: ' or "^" '
91 Injection or "^" failed
92 Testing payload: ' or "*" '
93 Injection or "*" failed
94 Testing payload: 'or true-- '
95 Injection or true-- failed
96 Testing payload: '" or true-- '
97 Injection " or true-- failed
98 Testing payload: "' or true--"
99 Injection ' or true-- failed
100 Testing payload: '") or true-- '
101 Injection ") or true-- failed
102 Testing payload: "') or true--"
103 Injection ') or true-- failed
104 Testing payload: "' or 'x'='x"
105 Injection ' or 'x'='x failed
106 Testing payload: "') or ('x')=('x"
107 Injection ') or ('x')=('x failed
108 Testing payload: '')) or (('x'))=(('x"
109 Injection ')) or (('x'))=(('x failed
110 Testing payload: '" or "x"="x'
111 Injection " or "x"="x failed
112 Testing payload: '") or ("x")=("x'
113 Injection ") or ("x")=("x failed
114 Testing payload: '")) or (("x"))=(("x'
115 Injection ") or (("x"))=(("x failed
116 Testing payload: 'or 1=1'
117 Injection or 1=1 failed
118 Testing payload: 'or 1=1-- '
119 Injection or 1=1-- failed
120 Testing payload: 'or 1=1#'
121 Injection or 1=1# failed
122 Testing payload: 'or 1=1/*'

```

```

123 Injection or 1=1/* failed
124 Testing payload: "admin' --"
125 Injection admin' -- failed
126 Testing payload: "admin' #"
127 Injection admin' # failed
128 Testing payload: "admin'/*"
129 Injection admin'/* failed
130 Testing payload: "admin' or '1'='1"
131 Injection admin' or '1'='1 failed
132 Testing payload: "admin' or '1'='1'--"
133 Injection admin' or '1'='1'-- failed
134 Testing payload: "admin' or '1'='1'#"
135 Injection admin' or '1'='1'# failed
136 Testing payload: "admin' or '1'='1'/*"
137 Injection admin' or '1'='1'/* failed
138 Testing payload: "admin' or 1=1 or ''='"
139 Injection admin' or 1=1 or ''=' failed
140 Testing payload: "admin' or 1=1"
141 Injection admin' or 1=1 failed
142 Testing payload: "admin' or 1=1--"
143 Injection admin' or 1=1-- failed
144 Testing payload: "admin' or 1=1#"
145 Injection admin' or 1=1# failed
146 Testing payload: "admin' or 1=1/*"
147 Injection admin' or 1=1/* failed
148 Testing payload: "admin') or ('1'='1"
149 Injection admin') or ('1'='1 failed
150 Testing payload: "admin') or ('1'='1'--"
151 Injection admin') or ('1'='1'-- failed
152 Testing payload: "admin') or ('1'='1'#"
153 Injection admin') or ('1'='1'# failed
154 Testing payload: "admin') or ('1'='1'/*"
155 Injection admin') or ('1'='1'/* failed
156 Testing payload: "admin') or '1'='1"
157 Injection admin') or '1'='1 failed
158 Testing payload: "admin') or '1'='1'--"
159 Injection admin') or '1'='1'-- failed
160 Testing payload: "admin') or '1'='1'#"
161 Injection admin') or '1'='1'# failed
162 Testing payload: "admin') or '1'='1'/*"
163 Injection admin') or '1'='1'/* failed
164 Testing payload: "1234 ' AND 1=0 UNION ALL SELECT 'admin', '81
    ↳ dc9bdb52d04dc20036dbd8313ed055"
165 Injection 1234 ' AND 1=0 UNION ALL SELECT 'admin', '81
    ↳ dc9bdb52d04dc20036dbd8313ed055 failed
166 Testing payload: 'admin" --'
167 Injection admin" -- failed
168 Testing payload: 'admin" #'
169 Injection admin" # failed
170 Testing payload: 'admin'/*'
171 Injection admin'/* failed
172 Testing payload: 'admin" or "1"="1'
173 Injection admin" or "1"="1 failed
174 Testing payload: 'admin" or "1"="1"--'
175 Injection admin" or "1"="1"-- failed
176 Testing payload: 'admin" or "1"="1"#'
177 Injection admin" or "1"="1"# failed
178 Testing payload: 'admin" or "1"="1'/*'
179 Injection admin" or "1"="1'/* failed
180 Testing payload: 'admin" or 1=1 or ""=""'
181 Injection admin" or 1=1 or ""="" failed
182 Testing payload: 'admin" or 1=1'

```

```

183 Injection admin" or 1=1 failed
184 Testing payload: 'admin" or 1=1--'
185 Injection admin" or 1=1-- failed
186 Testing payload: 'admin" or 1=1#'
187 Injection admin" or 1=1# failed
188 Testing payload: 'admin" or 1=1/*'
189 Injection admin" or 1=1/* failed
190 Testing payload: 'admin") or ("1"="1'
191 Injection admin") or ("1"="1 failed
192 Testing payload: 'admin") or ("1"="1"--'
193 Injection admin") or ("1"="1"-- failed
194 Testing payload: 'admin") or ("1"="1"#'
195 Injection admin") or ("1"="1"# failed
196 Testing payload: 'admin") or ("1"="1/*'
197 Injection admin") or ("1"="1/* failed
198 Testing payload: 'admin") or "1"="1'
199 Injection admin") or "1"="1 failed
200 Testing payload: 'admin") or "1"="1"--'
201 Injection admin") or "1"="1"-- failed
202 Testing payload: 'admin") or "1"="1"#'
203 Injection admin") or "1"="1"# failed
204 Testing payload: 'admin") or "1"="1/*'
205 Injection admin") or "1"="1/* failed
206 Testing payload: '1234 " AND 1=0 UNION ALL SELECT "admin", "81
    ↳ dc9bdb52d04dc20036dbd8313ed055'
207 Injection 1234 " AND 1=0 UNION ALL SELECT "admin", "81
    ↳ dc9bdb52d04dc20036dbd8313ed055 failed

```

LISTING 15. SQL injection script output

REFERENCES

- [1] payloadbox, *sql-injection-payload-list*, Oct. 2025. [Online]. Available: <https://github.com/payloadbox/sql-injection-payload-list>.
- [2] MyBatis, *MyBatis 3 | Introduction – mybatis*, Jan. 2025. [Online]. Available: <https://mybatis.org/mybatis-3>.
- [3] J. Hooke, *jBCrypt - strong password hashing for Java*, [Online; accessed 31. Oct. 2025], Jan. 2015. [Online]. Available: <https://www.mindrot.org/projects/jBCrypt>.
- [4] GeeksforGeeks, “Advanced Encryption Standard (AES),” *GeeksforGeeks*, Aug. 2025. [Online]. Available: <https://www.geeksforgeeks.org/computer-networks/advanced-encryption-standard-aes>.