Example #1

```
app.get('/profile/:userId', (req, res) => {
    User.findById(req.params.userId, (err, user) => {
        if (err) return res.status(500).send(err);
        res.json(user);
    });
});
```

The immediate issue with this particular cut of code is that the code requests an user ID but does not verify in the user using the user ID is authorized to use it due to the lack of verification. As such, an attacker can cause any number of data leaks and liability issues for the app's developer/corporation. As such, the best fix we be something like the following.

```
app.get('/profile/:userId', authenticateUser, (req, res) => {
    if (req.user.id !== req.params.userId) {
        return res.status(403).send('Access denied');
    }

    User.findById(req.params.userId, (err, user) => {
        if (err) return res.status(500).send(err);
        res.json(user);
    });
});
```

In this revised version of the code, the **authenticateUser** portion populates the user ID into the **req.user.id** variable to be later compared to the **req.params.userId** variable. If the two don't match then the user will be denied access.

Example #2

```
@app.route('/account/<user_id>')
def get_account(user_id):
    user = db.query(User).filter_by(id=user_id).first()
    return jsonify(user.to_dict())
```

Similar to the previous version, this code snippet fails to verify if the user trying to access the user data is the same user that the ID is attached to. The main difference is that this is via a URL. This carries the same liabilities and risks as the previous code, and as such should be fixed in a similar way.
Since this code is meant for a website, I aimed to perform a login verification. So right off the

```
@app.route('/account/<user_id>')
@login_required
def get_account(user_id):
    if str(current_user.id) != user_id:
        return jsonify({'error': 'Access denied'}), 403

    user = db.query(User).filter_by(id=user_id).first()
    if not user:
        return jsonify({'error': 'User not found'}), 404

    return jsonify(user.to_dict())
```

bat **@login_required** calls for a login to happen and will redirect to a login screen. This login will capture the user ID into the **current_user.id** variable and will then use that to compare it with the user Id that is being accessed. There is also a path for when the user ID is being logged into is not a user at all.

Example #3

```
1    public String hashPassword(String password) throws NoSuchAlgorithmException {
2        MessageDigest md = MessageDigest.getInstance("MD5");
3        md.update(password.getBytes());
4        byte[] digest = md.digest();
5        return DatatypeConverter.printHexBinary(digest);
6    }
```

Had this been the early 1990's this code would have been perfectly fine, but hindsight is 2020 and times have certainly changed. The issue for this particular snippet of code is the fact it is running an outdated and obsolete hashing system (MD5). This hashing system was used for passwords in the past, but has now fallen out of favor. Mainly do to the fact that it has a chance of producing the same hash for different entities, the hashes can be brute forced and with little time wasted. On top of that, there isn't any salting of the hash. So honestly the best option would be to create the snippet of code from scratch using a different hash entirely.

```
1    import de.mkammerer.argon2.Argon2Factory;
2    import de.mkammerer.argon2.Argon2;
3
4    public class PasswordHasher {
5        public String hashPassword(String password) {
6            Argon2 argon2 = Argon2Factory.create();
7
8            int iterations = 3;
9            int memory = 65536; // 64 MB
10           int parallelism = 2;
11
12           try {
13               return argon2.hash(iterations, memory, parallelism, password.toCharArray());
14           } finally {
15               argon2.wipeArray(password.toCharArray());
16           }
17       }
18
19       public boolean verifyPassword(String hash, String password) {
20           Argon2 argon2 = Argon2Factory.create();
21           return argon2.verify(hash, password.toCharArray());
22       }
23   }
```

Here I used a more updated and current hashing method that has salting and iterations built in (Argon2). It also has a memory wipe to ensure that after the hash is created, it is not stored with memory.

Example #4

```
1    import hashlib
2
3    def hash_password(password):
4        return hashlib.sha1(password.encode()).hexdigest()
```

Another outdated hashing function that does not salt the hash either. So in a similar fashion we have a script that uses speed to create a simple hash but is also vulnerable to attackers brute forcing with that very same speed. So we will do the same this and recreate the script using Argon2 and make it so that the hash is safer, salted, and memory dense for any attackers trying to brute force the hash.

```
from argon2 import PasswordHasher

ph = PasswordHasher()

def hash_password(password):
    return ph.hash(password)

def verify_password(hashed, password):
    try:
        return ph.verify(hashed, password)
    except:
        return False
```

With this the password has to go through a verification process, wasting more resources on the attacker, on each attempt to brute force.

Example #5

```
String username = request.getParameter("username");
String query = "SELECT * FROM users WHERE username = '" + username + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

This code's problem is that it does not prevent SQL injection attacks leading to possible manipulation of data and the acquisition of data that the attacker should not have access to. So if we are to repair it, we must address that vulnerability.

```
String username = request.getParameter("username");
String query = "SELECT * FROM users WHERE username = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, username);
ResultSet rs = pstmt.executeQuery();
```

Here we set up a better placeholder for the query so that the input is read as data and not as function or some form of SQL logic. Meaning even if the input was a form of SQL logic, it will simply get read as text data via the **setString()** function.

Example #6

```
app.get('/user', (req, res) => {
    // Directly trusting query parameters can lead to NoSQL injection
    db.collection('users').findOne({ username: req.query.username }, (err, user) => {
        if (err) throw err;
        res.json(user);
    });
});
```

So this is a similar form of injection known as NoSQL injection. It functions the same way of forcing a logic command into the query to be read in the database and manipulate it; except instead of SQL logic commands, they injection JSON logic commands. This found more often with systems like MongoDB, which is an example of a database not based in SQL; hence the name NoSQL. So much like the previous example, we need to make it so that the inputs are validated and read only as data.

```
1   const express = require('express');
2   const mongoSanitize = require('mongo-sanitize');
3   const app = express();
4   const db = require('./db');
5
6   app.get('/user', (req, res) => {
7       const rawUsername = req.query.username;
8
9       if (typeof rawUsername !== 'string' || !/^[a-zA-Z0-9_]{3,30}$/.test(rawUsername)) {
10          return res.status(400).json({ error: 'Invalid username format' });
11      }
12
13      const username = mongoSanitize(rawUsername);
14
15      db.collection('users').findOne({ username: username }, (err, user) => {
16          if (err) {
17              console.error('Database error:', err);
18              return res.status(500).json({ error: 'Internal server error' });
19          }
20
21          if (!user) {
22              return res.status(404).json({ error: 'User not found' });
23          }
24
25          res.json(user);
26      });
```

Definitely a big one. So here we are doing a few different things. For starters, we validated that only string based text was entered by using Line 9. This is to prevent any objects or arrays from being added. Secondly, on Line 13 we sanitize the username input to keep and NoSQL injection logic from being entered, since the $ is used in those logic injections and is considered safe string script. Thirdly we structured the query using Line 15 so that the database can only return what has been added in username variable (and subsequently passed the previous vetting tests).

Example #7

```
1   @app.route('/reset-password', methods=['POST'])
2   def reset_password():
3       email = request.form['email']
4       new_password = request.form['new_password']
5       user = User.query.filter_by(email=email).first()
6       user.password = new_password
7       db.session.commit()
8       return 'Password reset'
```

Well this is certainly a way of resetting a password. So we don't authenticate if the user is authorized to change the password, we don't hash the new password (and assumably didn't hash the old one either), and we aren't even checking the input for injections from SQL/NoSQL logic. It also will just break if the email isn't in the database to begin with. Which should be a none issue since to change a password you must first have created the password in the first place, which assumes you gave an email to use as a username. However, there is no check to make sure that the input for the email matches the email the password we are changing is attached to; which will create a logic error that breaks the app. So let's fix all of these problems.

```
1   from flask import request, jsonify
2   from argon2 import PasswordHasher
3   from itsdangerous import URLSafeTimedSerializer, BadSignature, SignatureExpired
4
5   ph = PasswordHasher()
6   serializer = URLSafeTimedSerializer('your-secret-key')  # Use a secure secret key
7
8   @app.route('/reset-password', methods=['POST'])
9   def reset_password():
10      token = request.form.get('token')
11      new_password = request.form.get('new_password')
12
13      if not new_password or len(new_password) < 8:
14          return jsonify({'error': 'Invalid password'}), 400
15
16      try:
17          email = serializer.loads(token, max_age=3600)  # Token expires in 1 hour
18      except SignatureExpired:
19          return jsonify({'error': 'Token expired'}), 403
20      except BadSignature:
21          return jsonify({'error': 'Invalid token'}), 403
22
23      user = User.query.filter_by(email=email).first()
24      if not user:
25          return jsonify({'error': 'User not found'}), 404
26
27      user.password = ph.hash(new_password)
28      db.session.commit()
29
30      return jsonify({'message': 'Password reset successful'})
```

Another big boy, but we had a lot to fix so let's go through it. First off, we now have verification of the user by sending a session token to the users' registered email as a second form of authentication. We can't assume that the current user has the password since the appropriate user also may not have it since they need to change it. Secondly, we added hashing via Argon2 (the sponsor of today's assignment apparently). Third, we added formatting to the password in Lines 13/14. Finally, we added various error states during the process of resetting the password; with corresponding error codes to go with them.

Example #8

```
<script src="https://cdn.example.com/lib.js"></script>
```

A super clear cut case here. We are just loading up external JavaScript without any vetting of said script. I'm sure I don't have to emphasize how dangerous this can get in the hands of an attacker. And when done automatically when loading up a website it becomes so much worse because the amount of things you can call upon is so vast that saying that this can completely destroy your system is reasonably apt. So let's add some vetting to this process.

```
<script src="https://cdn.example.com/lib.js"
        integrity="sha384-abc123..."
        crossorigin="anonymous"></script>
```

There we go. Adding some Subresource Integrity check will ensure that your browser will only execute the third party JavaScript if it is the JavaScript originally designated by the creator.

Example #9

```
url = input("Enter URL: ")
response = requests.get(url)
print(response.text)
```

Seems pretty simple. You give it a URL, it creates a request to the server using that URL, and then provides an output based on that request. One question though, why is there no authentication or no vetting of inputs? This can lead, once again, to many data leaks from attackers having access to data they aren't suppose to have access to. Time to fix that.

```
import requests
from urllib.parse import urlparse
import socket
import ipaddress

ALLOWED_HOSTS = ['api.example.com', 'data.example.org']
TIMEOUT = 5  # seconds

def is_private_ip(hostname):
    try:
        ip = socket.gethostbyname(hostname)
        return ipaddress.ip_address(ip).is_private
    except Exception:
        return True  # Treat resolution failure as unsafe

def safe_request(url):
    parsed = urlparse(url)

    # 1. Validate scheme
    if parsed.scheme not in ['http', 'https']:
        raise ValueError("Invalid URL scheme. Only HTTP and HTTPS are allowed.")

    # 2. Validate hostname against whitelist
    if parsed.hostname not in ALLOWED_HOSTS:
        raise ValueError("Untrusted destination host.")

    # 3. Block internal/private IPs
    if is_private_ip(parsed.hostname):
        raise ValueError("Request blocked: private or internal IP detected.")

    # 4. Make the request safely
    try:
        response = requests.get(url, timeout=TIMEOUT)
        response.raise_for_status()
        return response.text
    except requests.exceptions.RequestException as e:
        raise RuntimeError(f"Request failed: {e}")
```

We tackle this pair of issues from three points. The first point is by whitelisting URLs in a list of AllowedHosts (see Line 24). Inversely, our second point of solving this issue is to blacklist all private IP addresses to prevent malicious inquiries (See Line 28). Finally, our third point of defense is to run URLs that are neither whitelisted or blacklisted through a controlled service scheme to test the URL for various errors (See Line 20).

Example #10

```
1     if (inputPassword.equals(user.getPassword())) {
2         // Login success
3     }
```

This is pretty much returning to the same problems we were seeing in Example #3 and #4. If the password given equals the password stored, then boom login success. Except, why is the password written and stored locally; in plain text no less? No hashing is seen, no brute forcing prevention is established, and not even a password format validation system. Let's fix these things.

```
1     import de.mkammerer.argon2.Argon2;
2     import de.mkammerer.argon2.Argon2Factory;
3
4     public class AuthService {
5         private static final Argon2 argon2 = Argon2Factory.create();
6
7         public boolean verifyLogin(String inputPassword, String storedHash) {
8             if (inputPassword == null || storedHash == null) {
9                 return false;
10            }
11
12            try {
13                return argon2.verify(storedHash, inputPassword.toCharArray());
14            } catch (Exception e) {
15                // Log error securely
16                return false;
17            }
18        }
19    }
```

So let's start with the sponsor in the room. Argon2 was used to hash the passwords so that they aren't stored in plain text. We also automatically salt it so the brute forcing will be much harder to even do. Secondly, Argon2 can be used to validate the stored password with the input one on Line 13. Speaking of Line 13, it is wrapped in a **trycatch** which will help prevent any errors or crashes born from those errors.