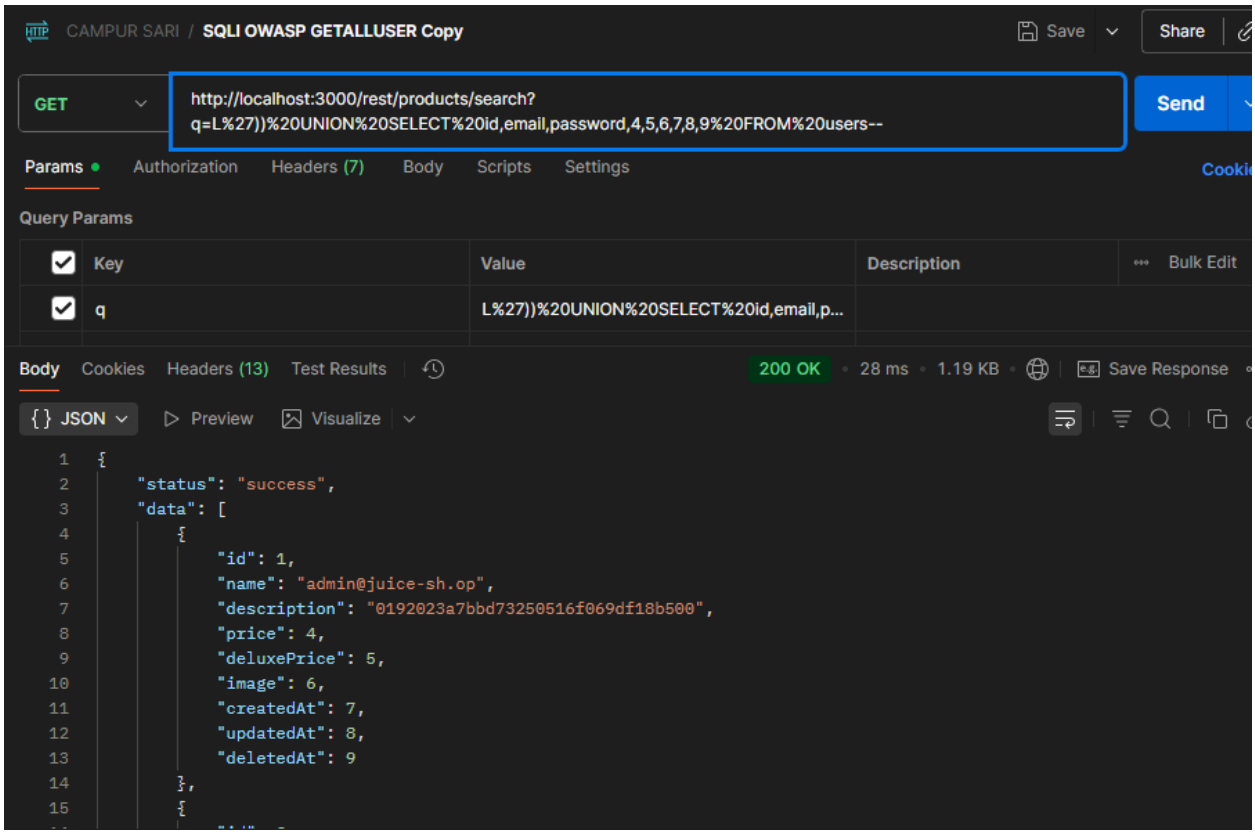
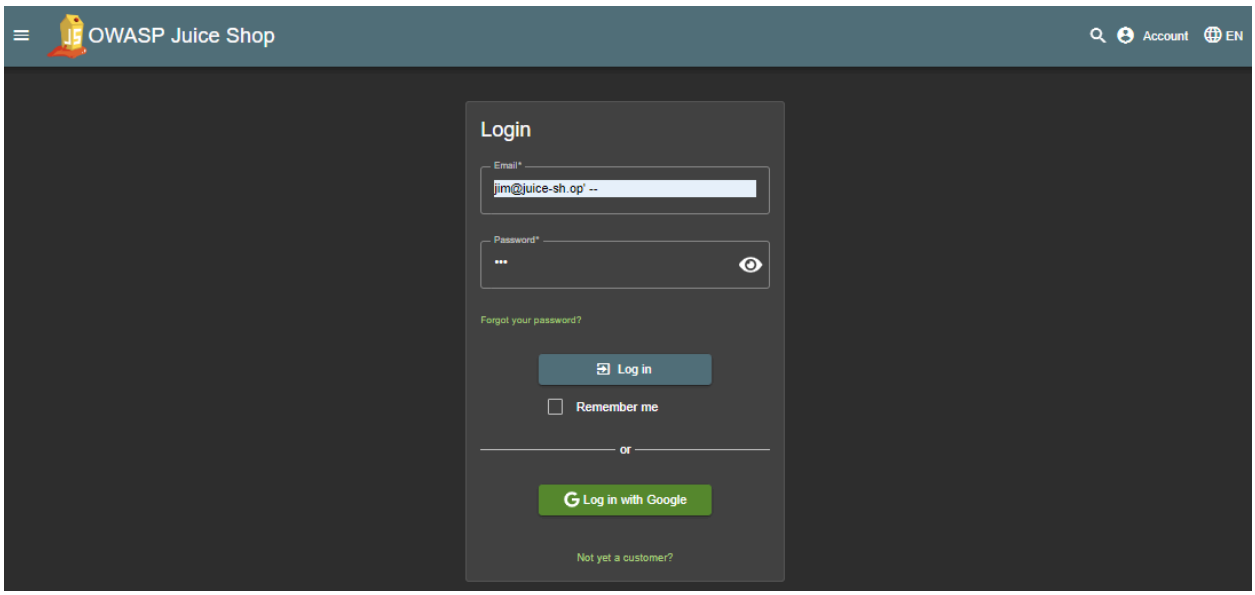
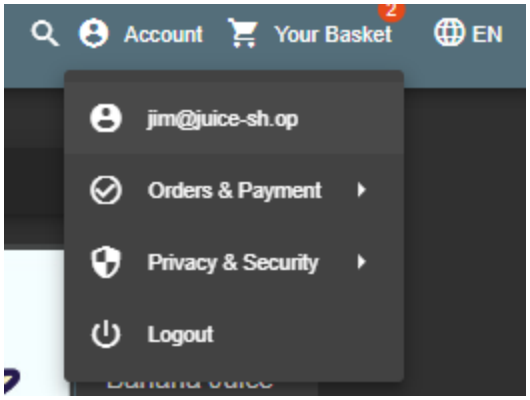
TASK A - SQLI

1. **User Credentials**
   Retrieve a list of all user credentials via SQL Injection.



2. **Log in with Jim's user account.**





Beberapa alasan mengapa ini rentan dan bagaimana mencegahnya (cheatsheetseries.owasp.org):

- Query dibentuk dengan penggabungan string – Memasukkan input user ke dalam perintah SQL tanpa validasi membuat database menjalankan kode yang tidak seharusnya

- Kurangnya pembatasan hak – Jika akun database yang dipakai aplikasi memiliki hak baca penuh ke tabel user, hasil `UNION SELECT` dari tabel `users` dapat diproses. Penerapan prinsip *least privilege* dapat membatasi dampak serangan.

- Tidak ada validasi/allow-list – Aplikasi tidak memeriksa apakah parameter `q` hanya berisi huruf dan angka yang diharapkan, sehingga karakter khusus seperti tanda petik (‘) dapat lolos.

Solusinya:

1. Gunakan *prepared statements* (parameterized queries) – Daripada membangun query dengan string + input, buat perintah SQL dengan placeholder lalu ikat nilai input sebagai parameter. Teknik ini memisahkan kode SQL dari data, sehingga input tidak pernah dianggap bagian dari query

2. Validasi dan sanitasi input – Terapkan *allow-list* yang hanya mengizinkan pola karakter tertentu (mis. huruf, angka) untuk parameter pencarian. Ini mencegah karakter seperti `'`, `--` atau `;` yang sering dipakai dalam injeksi

3. Gunakan ORM atau stored procedure yang aman – Banyak framework (Sequelize, SQLAlchemy, Hibernate) menyediakan API yang secara otomatis membuat query terparameterisasi. Bila memakai stored procedure, pastikan prosedur dibangun dengan parameter yang terikat dan bukan menggabungkan string

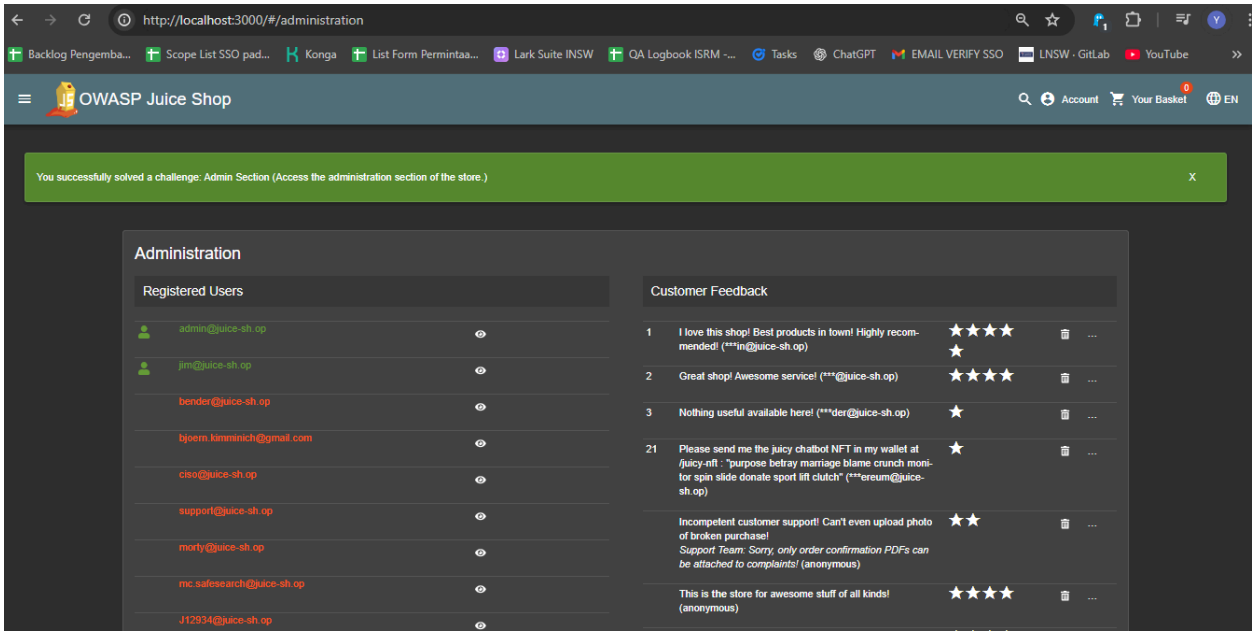
4. Minimalisir hak akses database – Beri aplikasi hanya hak yang diperlukan (mis. hanya SELECT untuk tabel tertentu). Jika terjadi injeksi, penyerang tidak bisa membaca tabel sensitif lainnya.

5. Audit dan tes keamanan – Secara berkala lakukan uji penetrasi dan tinjau kode untuk menemukan kueri dinamis berbahaya. Tools seperti linter atau scanner dapat membantu mendeteksi pola berisiko.

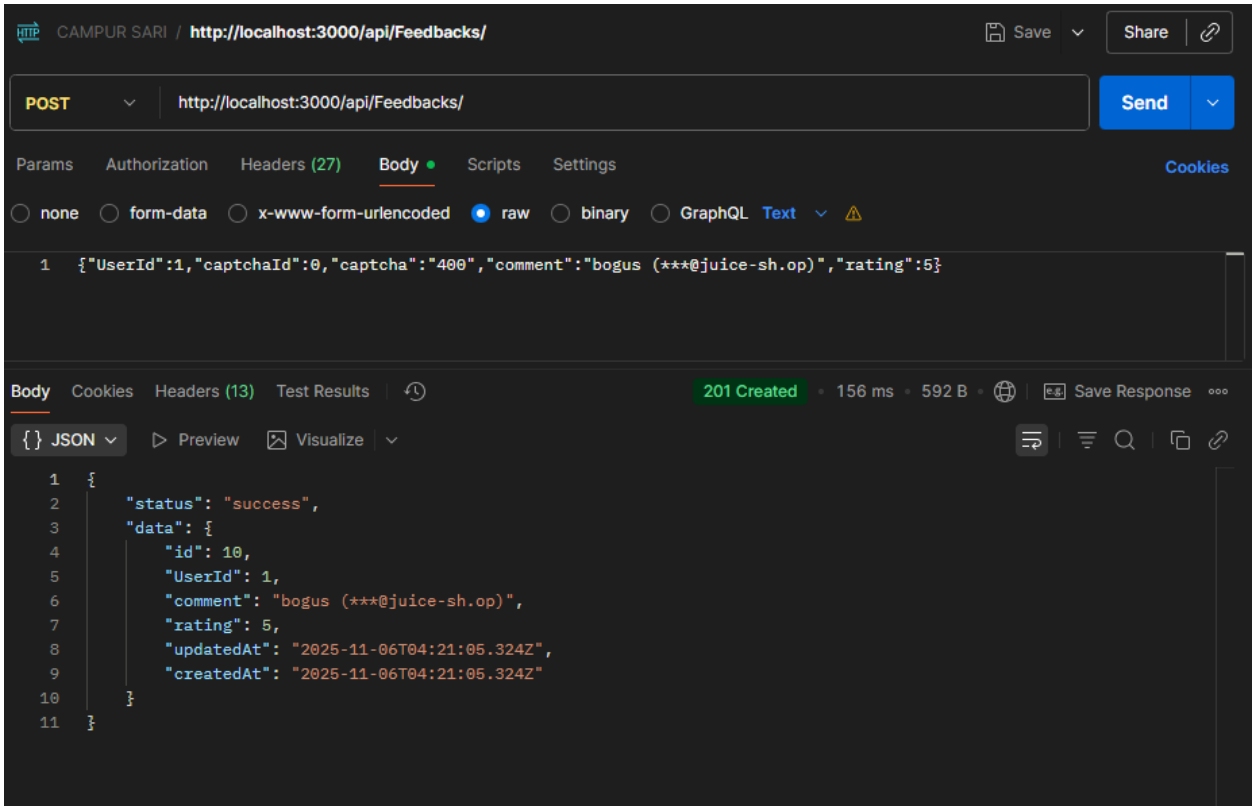
Dengan menerapkan teknik-teknik di atas, aplikasi tidak akan mengeksekusi perintah yang disisipkan oleh input user, sehingga serangan SQL Injection seperti pada challenge tersebut dapat dicegah.

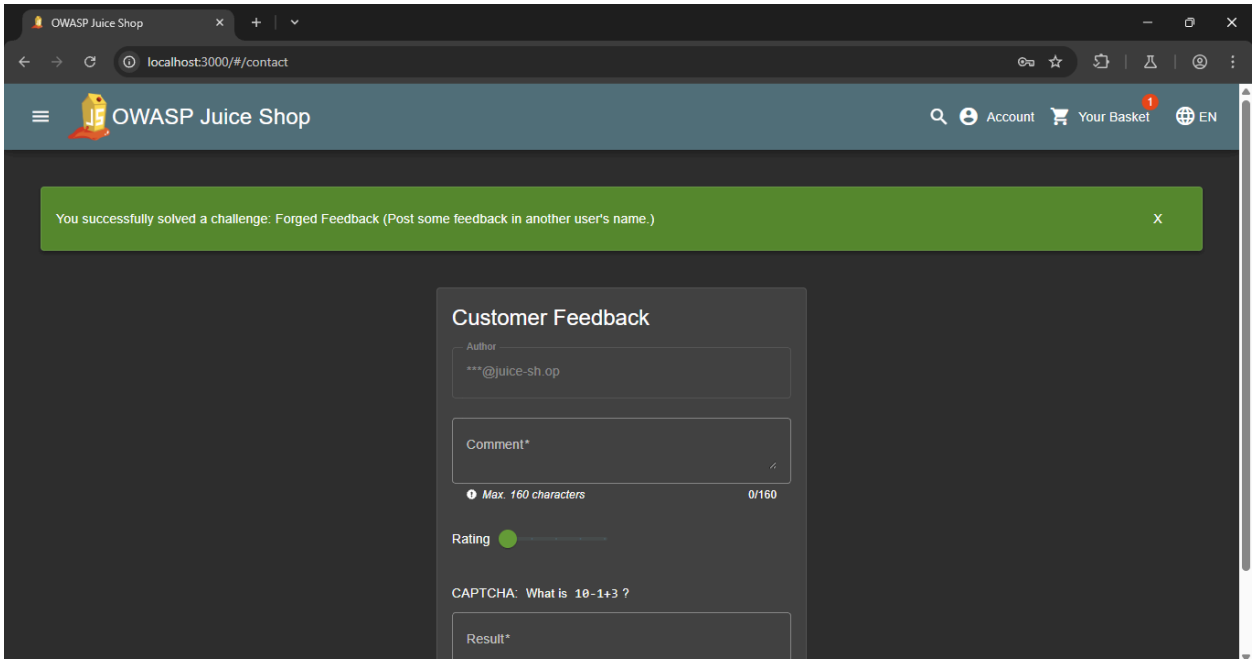
TASK B - BROKEN ACCESS CONTROL

1. **Access Admin Page**
   Ini sebenarnya hasil dari sql injection yang berhasil masuk ke page khusus admin



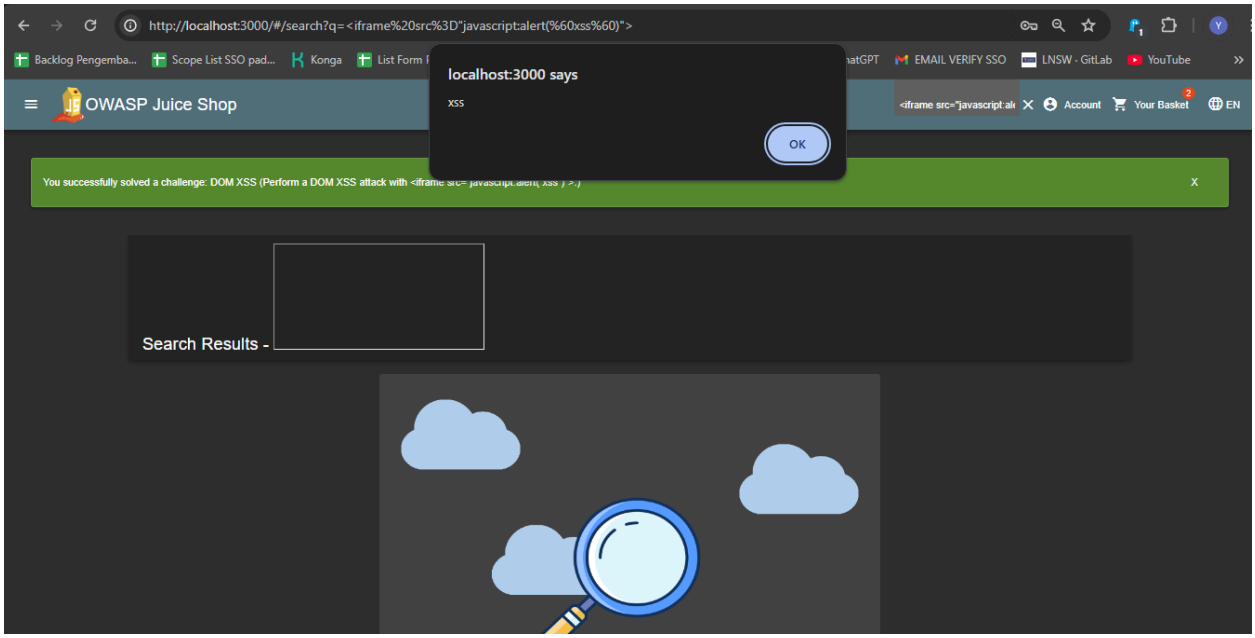
2. **Forged Feedback**
   Post some feedback in another user's name.Dapat dengan mudah mengirim payload dan mengganti id user pada payload sehingga feedback dikirim dengan user id lain
   - Karena id mudah ditebak
   - Tidak melakukan validasi authorized user





TASK C - XSS VULNERABILITY ANALYSIS

1. **Perform a DOM XSS attack with <iframe src="javascript:alert(`xss`)">. Kenapa DOM-based XSS bisa terjadi (penjelasan sederhana)**



DOM-based XSS terjadi ketika JavaScript di sisi klien mengambil data dari sumber yang tidak tepercaya (mis. URL, fragment/hash, document.referrer, localStorage, postMessage) lalu menulis ulang DOM dengan data itu secara tidak aman — mis. menggunakan innerHTML, document.write, eval, atau menyetel atribut yang mengeksekusi kode. Karena semua berlangsung di browser, payload tidak perlu lewat server sehingga sering sulit dideteksi. (Referensi: OWASP DOM XSS & XSS Prevention).

Contoh alur singkat:

- Aplikasi baca location.hash atau parameter q dari URL.
- Kode JS menaruh isi itu ke someElement.innerHTML = userInput;
- Jika userInput berisi markup/javasript, browser akan mengeksekusinya → XSS.

Cara mencegah / solusi (praktis & prioritas)

Utamakan *defense in depth*: beberapa lapis proteksi, bukan satu solusi tunggal. (Saran berdasarkan OWASP cheat sheets)

1. **Hentikan penggunaan `innerHTML` / `document.write` / `eval` untuk memasukkan data user.**
    - Gunakan: `element.textContent` atau `element.innerText` untuk memasukkan teks (aman).

Jika perlu memasukkan elemen, buat elemen programatik:

```
const span = document.createElement('span');
span.textContent = userInput; // safe
container.appendChild(span);
```

-
- Untuk atribut: `element.setAttribute('href', safeUrl)`; namun verifikasi `safeUrl` terlebih dahulu.

2. **Sanitasi HTML jika memang perlu memasukkan markup**
    - Pakai library yang mature (mis. **DOMPurify**) untuk membersihkan HTML sebelum dimasukkan ke `innerHTML`. Jangan buat sanitizer custom sendiri tanpa pemahaman mendalam. DOMPurify
3. **Gunakan Trusted Types** (browser feature) bila memungkinkan — membatasi assignment ke sink-sensitive seperti `innerHTML` kecuali melalui API yang aman.
4. **Terapkan Content Security Policy (CSP)** sebagai lapisan tambahan.
    - CSP yang ketat (nonce/hash untuk inline script) membuat eksploit XSS jauh lebih sulit. Tapi CSP bukan substitusi untuk coding aman. OWASP Cheat Sheet Series
5. **Allow-list input** bila konteksnya terdefinisi (mis. parameter yang hanya boleh berupa kata/angka). Jangan andalkan blacklist. OWASP Cheat Sheet Series
6. **Validasi sumber data** — jangan percaya data dari `postMessage`, `referrer`, atau 3rd-party tanpa verifikasi. Pastikan domain/format sesuai ekspektasi.
7. **Frameworks & templating**
    - Gunakan engine/template yang otomatis melakukan escaping (React, Angular, Vue punya mekanisme auto-escape). Pahami celah bila menggunakan `dangerouslySetInnerHTML` di React, dsb.
8. **Code review & secure coding checklist**
    - Review semua penggunaan DOM sinks. Cari `innerHTML`, `insertAdjacentHTML`, `eval`, `setAttribute('src'|'href' ...)` dan pastikan ada validasi/sanitasi.

9. **Testing & scanning**
    - Tambahkan test otomatis (unit/integration) untuk memastikan input tertentu di-escape.
    - Gunakan scanning tools (SAST/DAST) dan manual review (pentest) untuk mendeteksi DOM XSS. Tools pembantu: Burp/Browser-based testers; lihat juga OWASP cheat sheets. [OWASP Cheat Sheet Series+1]
10. **Runtime monitoring**
    - Aktifkan CSP report-uri untuk menerima laporan serangan XSS di lapangan. Monitor console errors, anomaly pada behavior, dll. [OWASP Cheat Sheet Series]

2. Bonus Payload

Use the bonus payload `<iframe width="100%" height="166" scrolling="no" frameborder="no" allow="autoplay" src="https://w.soundcloud.com/player/?url=https%3A//api.soundcloud.com/tracks/771984076&color=%23ff5500&auto_play=true&hide_related=false&show_comments=true&show_user=true&show_reposts=false&show_teaser=true"></iframe>` in the DOM XSS challenge. (1 Star)