

Secure Coding Frontend Dev



Anhar Solehudin
Bandung, Agustus 2025





About Me



Anhar Solehudin

7+ Software Engineer

Tech Lead at Digital Service Digitization
Telkom Indonesia

Ex-Tech Lead CyberArmy Indonesia

Writing Secure Coding at harscode.dev

Portfolio



FAB DIGITAL



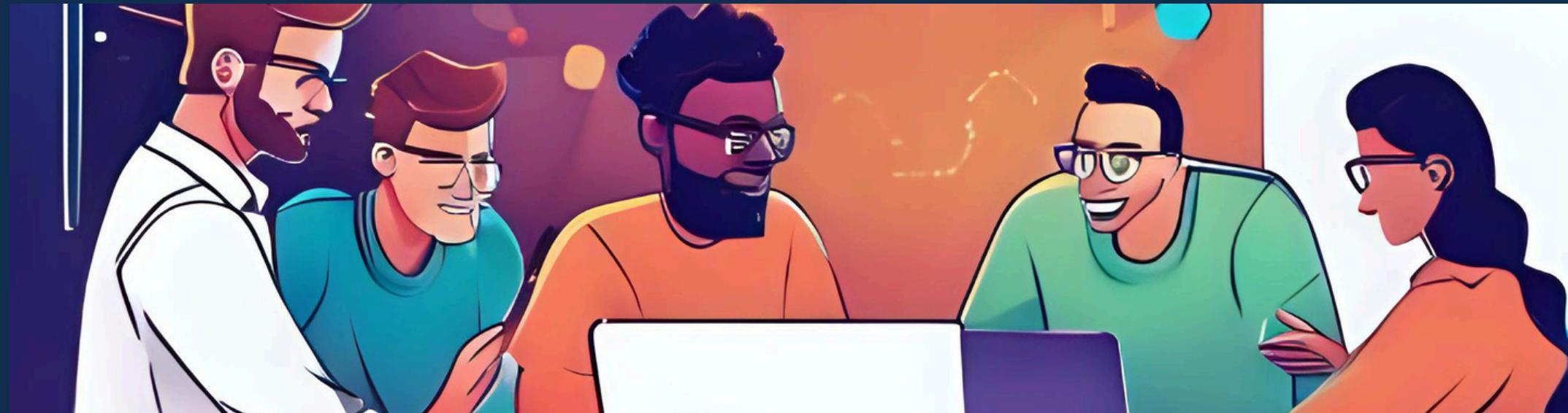
**Classified
Project**

Pre-Test

30 Minutes



What You'll Do



Explore Common
Security Bug in
Frontend Development
& VueJS

Secure Coding Best
Practices for Frontend
Development & VueJS

Patch the Bugs You
Discovered on
Tourist Arrival App

The Myth



Common Security Bug & Best Practices



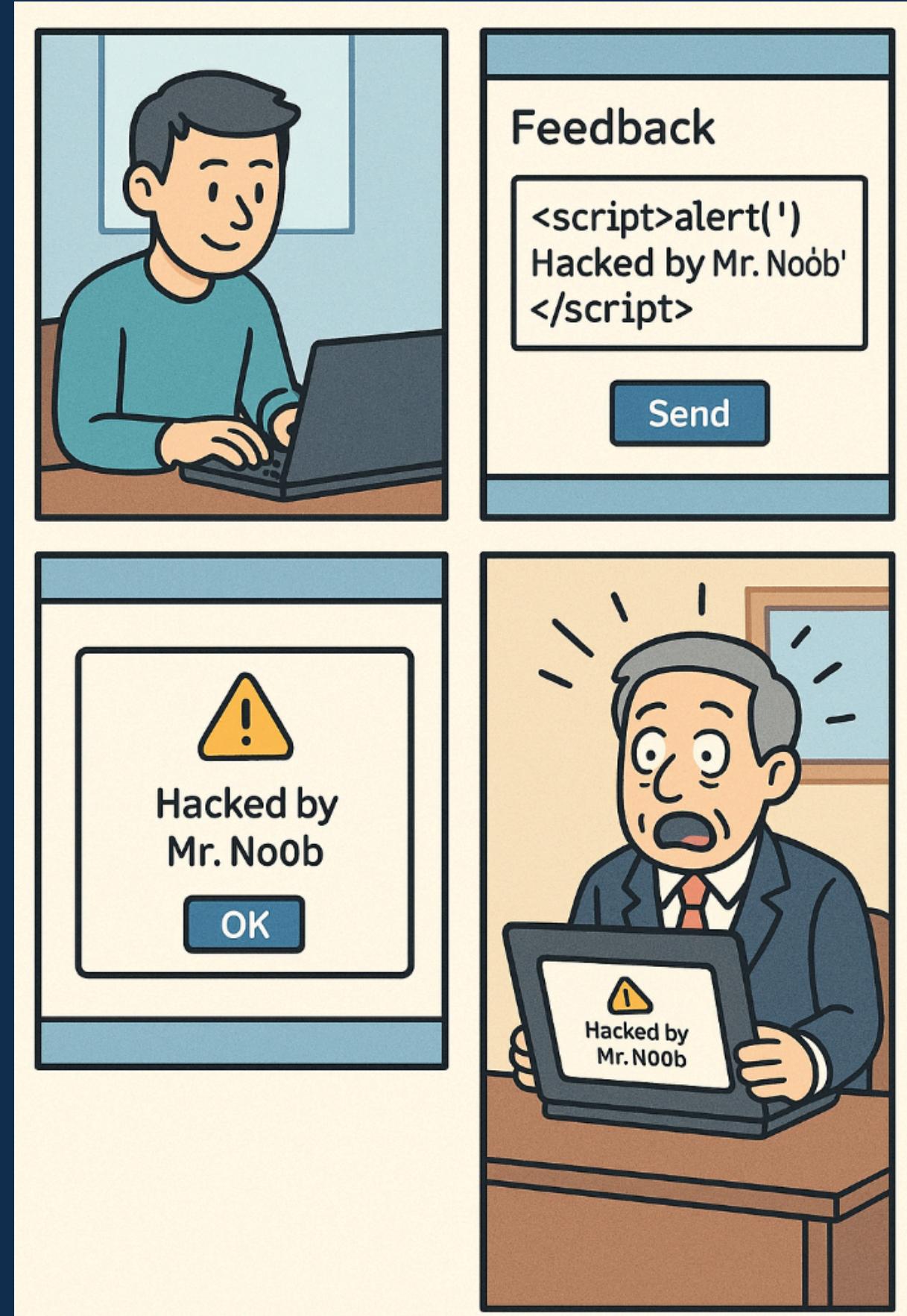
XSS (Cross Site Scripting) Script Kecil, Masalah Besar

Kamu bikin form feedback
sederhana.

Seminggu kemudian, CEO
buka dashboard, muncul
alert:

Hacked by Mr. No0b

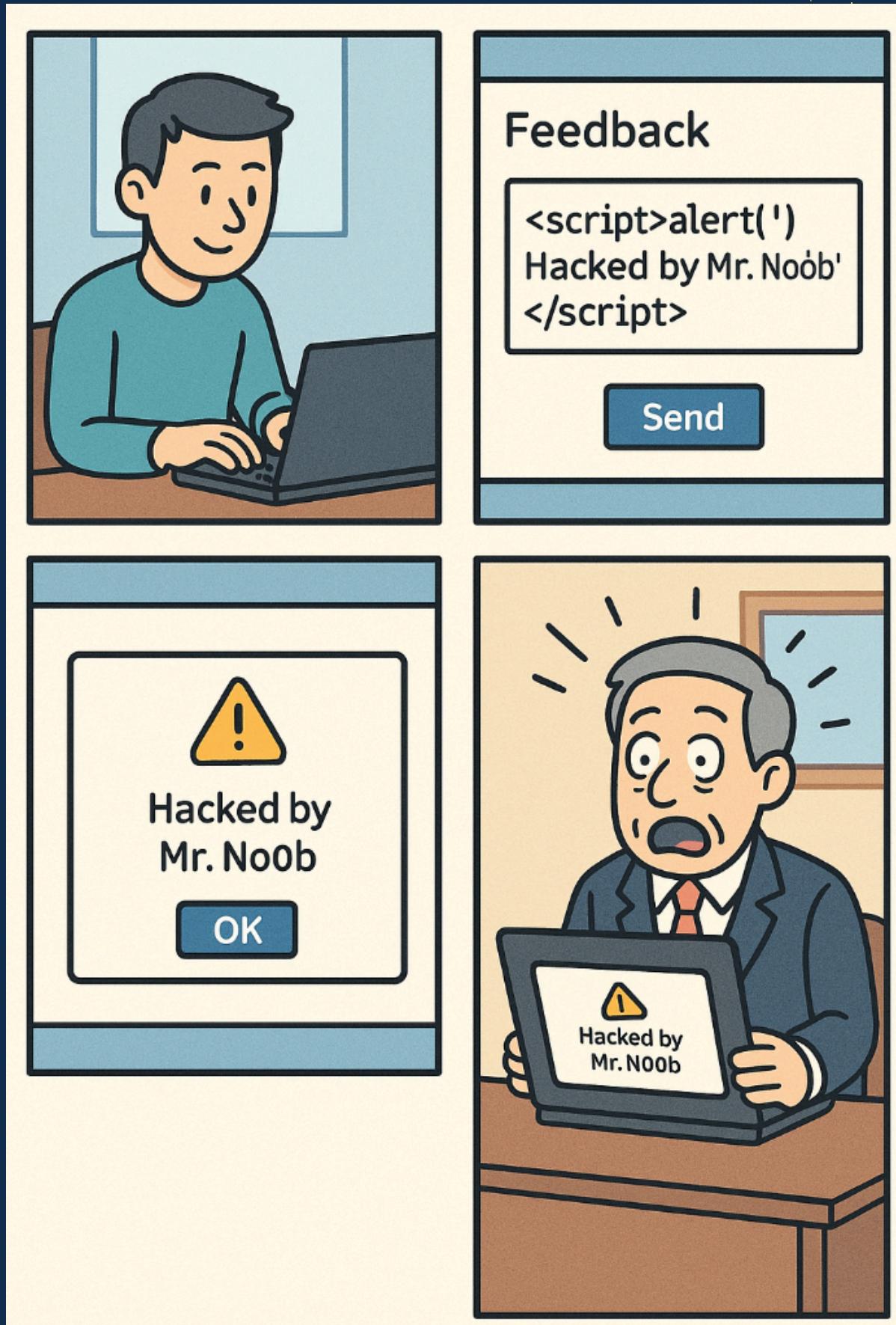
Dan Pak CEO pun Panik....



Root Cause
Developer langsung render input user ke DOM pakai innerHTML → script injeksi bisa jalan

Impact

Attacker bisa mencuri cookie/session, impersonasi user, eksekusi arbitrary script (mencuri data, injeksi malware, dsb).



Before Patch

```
<div id="feedback"></div>
<script>
  const feedback = new URLSearchParams(location.search).get('msg');
  document.getElementById('feedback').innerHTML = feedback; // ✗ raw HTML
</script>
```

After Patch

```
<div id="feedback"></div>
<script>
  const feedback = new URLSearchParams(location.search).get('msg');
  document.getElementById('feedback').textContent = feedback; // ✓ aman
</script>
```

Best Practices

- Jangan gunakan innerHTML untuk input user
- Gunakan.textContent atau Vue binding {{ variable }}
- Sanitasi HTML dengan DOMPurify jika butuh raw HTML
- Aktifkan Content Security Policy (CSP)
- Tambahkan unit test untuk input injection

QUIZ

Dari kode berikut, bagian mana yang membuka celah XSS? Bagaimana cara cepat memperbaikinya?

```
const name = new URLSearchParams(location.search).get('name');
document.getElementById('welcome').innerHTML = "Hi " + name;
```

- A. Menghapus kode tersebut
- B. Mengganti innerHTML ke textContent
- C. Menambah alert('Hacked') di script
- D. Menyimpan name di database

CSRF (Cross-Site Request Forgery)

- ○ ○
- ○ ○

Kamu bikin form ubah password, kelihatan sederhana.

Tapi tiba-tiba ada user yang nggak pernah login bisa ganti password orang lain hanya dengan klik link di email.



Aplikasi tidak memvalidasi apakah request berasal dari halaman asli

Root Cause

Attacker bisa ganti password user, hapus data, transaksi finansial tanpa sepengetahuan user.

Impact



Before Patch

```
<form action="/change-password" method="POST">
    <input type="password" name="newPassword" />
    <button type="submit">Change Password</button>
</form>
<!-- Tidak ada token CSRF -->
```

After Patch

```
<form action="/change-password" method="POST">
    <input type="hidden" name="csrfToken" value="{{ token }}" />
    <input type="password" name="newPassword" />
    <button type="submit">Change Password</button>
</form>
<!-- Server memverifikasi token CSRF -->
```

Best Practices

- Tambahkan Anti-CSRF Token pada setiap request yang mengubah data
- Validasi Origin/Referer header di server
- Gunakan SameSite Cookie (Lax atau Strict)
- Gunakan security feature framework (contoh: middleware CSRF bawaan)

QUIZ

Dari potongan form berikut, apa yang membuat aplikasi rentan CSRF?
Bagaimana perbaikannya?

```
<form action="/delete-account" method="POST">  
    <button type="submit">Delete My Account</button>  
</form>
```

- A. Tambahkan token CSRF dan verifikasi server
- B. Gunakan GET method saja
- C. Pindahkan form ke halaman lain
- D. Tidak perlu perubahan, sudah aman



IDOR (Insecure Direct Object Reference)

Kamu bikin checkout page: ?
product_id=1&price=500000.
Attacker buka DevTools,
ubah jadi price=5000.
Checkout sukses, barang
sampai, bisnis kamu tekor.



Backend percaya nilai harga yang dikirim dari client tanpa validasi.

Impact

- Attacker bisa membeli produk dengan harga tidak semestinya → kerugian finansial & reputasi hancur.
- Bisa diperluas: manipulasi quantity, diskon palsu, bahkan pembelian item yang seharusnya restricted.



Before Patch

```
// Harga diambil langsung dari parameter request
const price = req.body.price;
const productId = req.body.product_id;
db.insertOrder({ productId, price, userId: req.session.userId });
```

After Patch

```
// Harga diambil dari database, bukan dari input user
const product = db.getProductById(req.body.product_id);
const price = product.price;
db.insertOrder({ productId: product.id, price, userId: req.session.userId });
```

Best Practices

- Jangan percaya data penting (harga, diskon, role) yang dikirim dari frontend
- Ambil nilai kritis langsung dari server/database
- Validasi hak akses untuk setiap transaksi
- Logging & monitoring untuk transaksi anomali
- Tambahkan test untuk endpoint sensitif

QUIZ

Dari kode berikut, kenapa rentan IDOR dan bagaimana solusinya?

```
app.post('/checkout', (req, res) => {
  const order = {
    price: req.body.price,
    productId: req.body.product_id,
    userId: req.session.userId
  };
  db.saveOrder(order);
  res.send("Order success");
});
```

- A. Gunakan harga langsung dari parameter, sudah cukup
- B. Validasi harga dari database berdasarkan product_id
- C. Hapus field price dari table order
- D. Gunakan GET method untuk checkout



Clickjacking (Frame Embedding Attack)

Kamu bikin halaman transfer dana yang aman.

Attacker sematkan halamanmu di iframe transparan, kasih tampilan lucu seperti tombol ‘Dapatkan Hadiah’.

User klik tombol hadiah, padahal itu aslinya tombol transfer uang ke attacker.



-
-
-

Halaman aplikasi bisa di-embed di iframe pihak ketiga tanpa proteksi.

Root Cause

User tanpa sadar melakukan aksi berbahaya: transfer uang, ubah password, hapus akun.

Impact



Before Patch

```
<!-- Halaman sensitif bisa di-embed tanpa proteksi -->
<html>
  <head>
    <title>Transfer</title>
  </head>
  <body>
    <form action="/transfer" method="POST">
      <input type="text" name="to" value="attacker">
      <button type="submit">Transfer</button>
    </form>
  </body>
</html>
```

After Patch

```
# Tambahkan Header HTTP Anti-Frame
X-Frame-Options: DENY
# atau yang lebih modern:
Content-Security-Policy: frame-ancestors 'none';
```

Best Practices

- Tambahkan header X-Frame-Options: DENY atau SAMEORIGIN
- Gunakan CSP frame-ancestors 'none' untuk kontrol embed modern
- Hindari iframe untuk konten sensitif
- Audit third-party widget (iklan, plugin) yang menggunakan iframe



QUIZ

Dari opsi berikut, mana yang bisa mencegah Clickjacking?

- A. Menggunakan X-Frame-Options header
- B. Mengganti semua tombol dengan link biasa
- C. Menambah password yang lebih panjang
- D. Memblokir semua koneksi internet



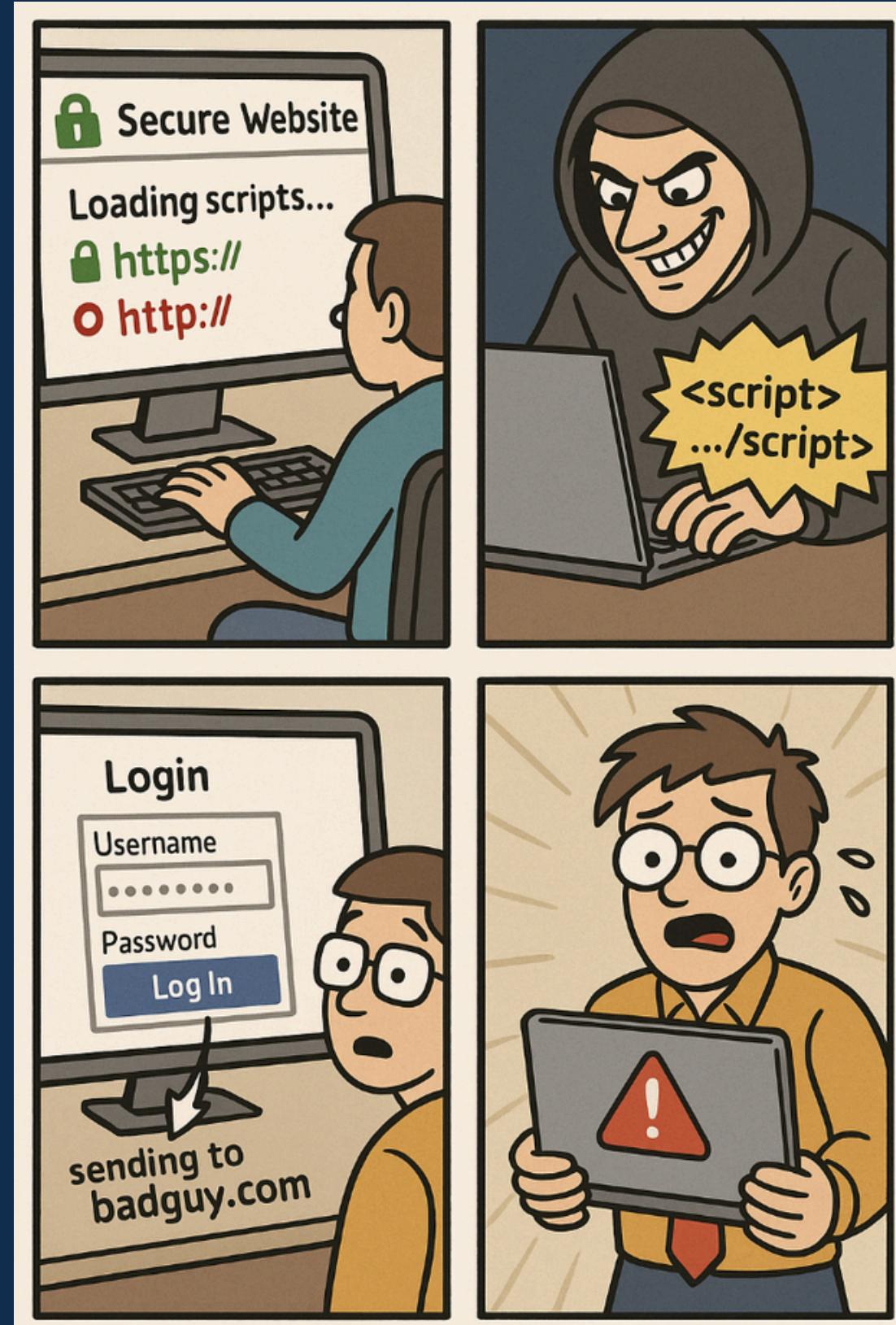
Mixed Content (HTTP vs HTTPS)

- ○ ○
- ○ ○

Aplikasi kamu sudah pakai
HTTPS, keren.

Tapi satu file script di-load
dari http://.

Attacker masuk di tengah
jalan, inject script sendiri,
tiba-tiba halaman login kirim
credential ke server lain.

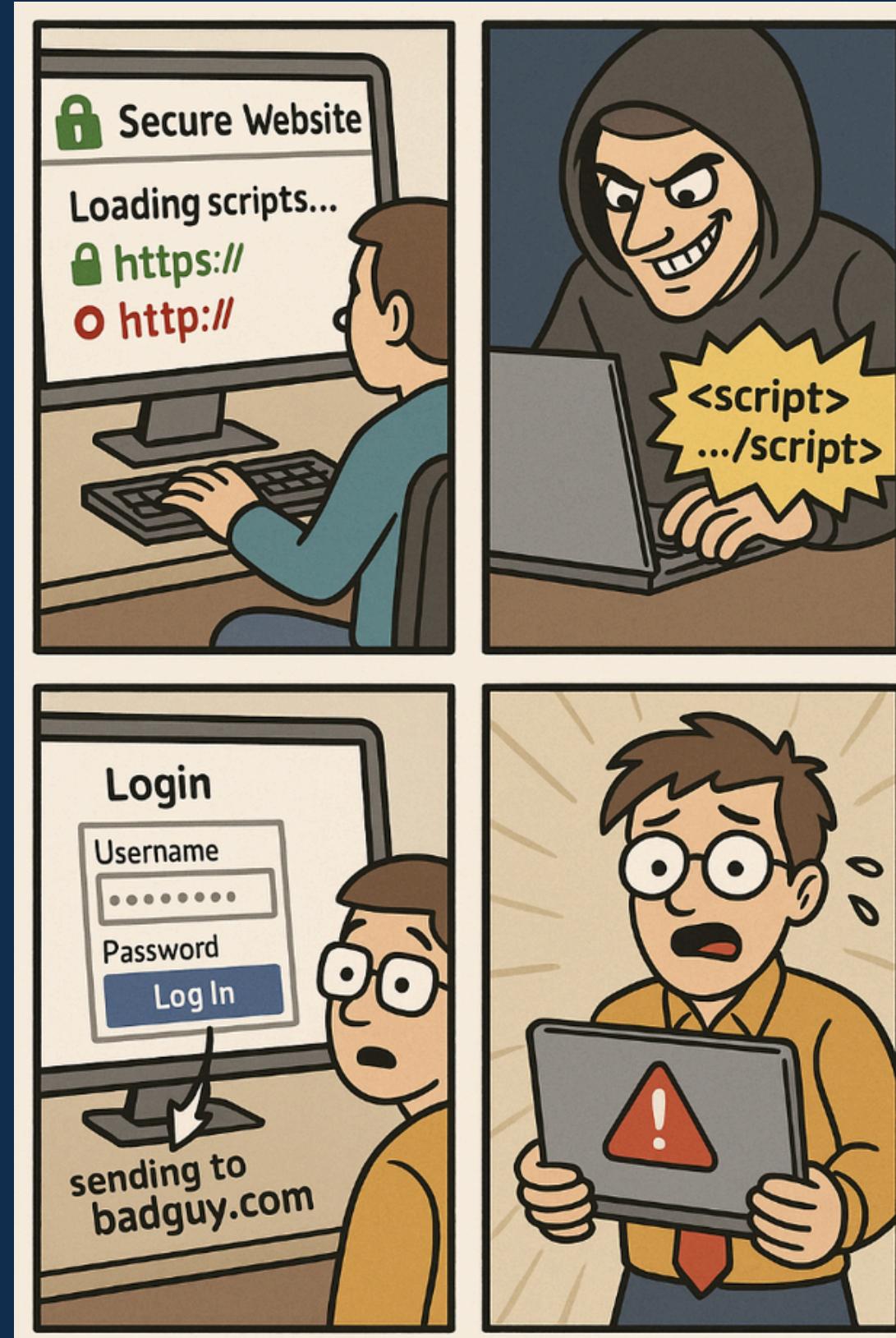


Root Cause

Memuat resource (JS, CSS, gambar) via HTTP pada halaman HTTPS.

Impact

Attacker bisa memanipulasi konten halaman: inject malware, sniff data user, menampilkan phishing.



Best Practices

- Pastikan semua resource di-load via HTTPS
- Gunakan Content Security Policy (CSP) upgrade-insecure-requests
- Audit dependency untuk memastikan tidak ada asset HTTP
- Hapus link absolut HTTP yang hardcoded di codebase

QUIZ

Apa risiko memuat resource http:// di halaman https://?

- A. Resource akan otomatis upgrade ke HTTPS
- B. Attacker bisa inject script berbahaya
- C. Tidak ada risiko, hanya beda port
- D. Halaman akan menjadi lebih cepat



CSP Missing / Salah Konfigurasi

-
-
-
-

Aplikasi kamu sudah rilis,
keren dan cepat.

Tapi ada satu bug kecil: XSS
berhasil inject script dari
domain aneh.

Kenapa bisa? Karena tidak
ada Content Security Policy
(CSP) yang melarangnya.



Root Cause

Tidak ada CSP, atau CSP longgar
(unsafe-inline, * untuk script).

Impact

- XSS lebih mudah dieksplorasi.
- Konten bisa diubah attacker (malware, keylogger, defacement).
- Sulit mendeteksi domain jahat yang menyisipkan script.



Best Practices

- Tambahkan header CSP yang ketat (default-src 'self')
- Hindari unsafe-inline dan unsafe-eval jika memungkinkan
- Gunakan nonce atau hash untuk inline script yang diperlukan
- Uji CSP dalam mode report-only sebelum diterapkan penuh

QUIZ

Kenapa CSP penting meskipun sudah menggunakan HTTPS?

- A. CSP mempercepat loading halaman
- B. CSP mencegah eksekusi script dari sumber tidak terpercaya
- C. CSP menggantikan kebutuhan validasi input
- D. CSP hanya untuk debugging

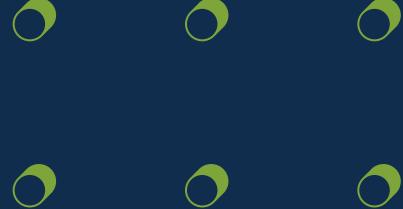


Sensitive Data Exposure (Local Storage & Session)

• • •
• • •

Kamu simpan access token di
localStorage biar gampang
dipakai.

Eh, suatu hari ada bug XSS
kecil, dan semua token user
dicuri.



Root Cause

Menyimpan token, credential, atau PII di tempat yang bisa diakses JavaScript (localStorage / sessionStorage).

Impact

- XSS = semua token bocor.
- Akun user bisa diambil alih, data sensitif bocor, reputasi rusak.

Before Patch

```
// Menyimpan token di localStorage
localStorage.setItem("accessToken", token);

// Membaca token dari localStorage di setiap request
const token = localStorage.getItem("accessToken");
```

After Patch

```
// Simpan token di HttpOnly Secure Cookie
res.cookie("accessToken", token, {
    httpOnly: true,
    secure: true,
    sameSite: "Strict"
});
```

Best Practices

- Jangan simpan credential di localStorage/sessionStorage
- Gunakan HttpOnly Secure Cookie untuk token
- Hapus data sensitif di browser saat logout
- Gunakan short-lived token + refresh token berbasis cookie

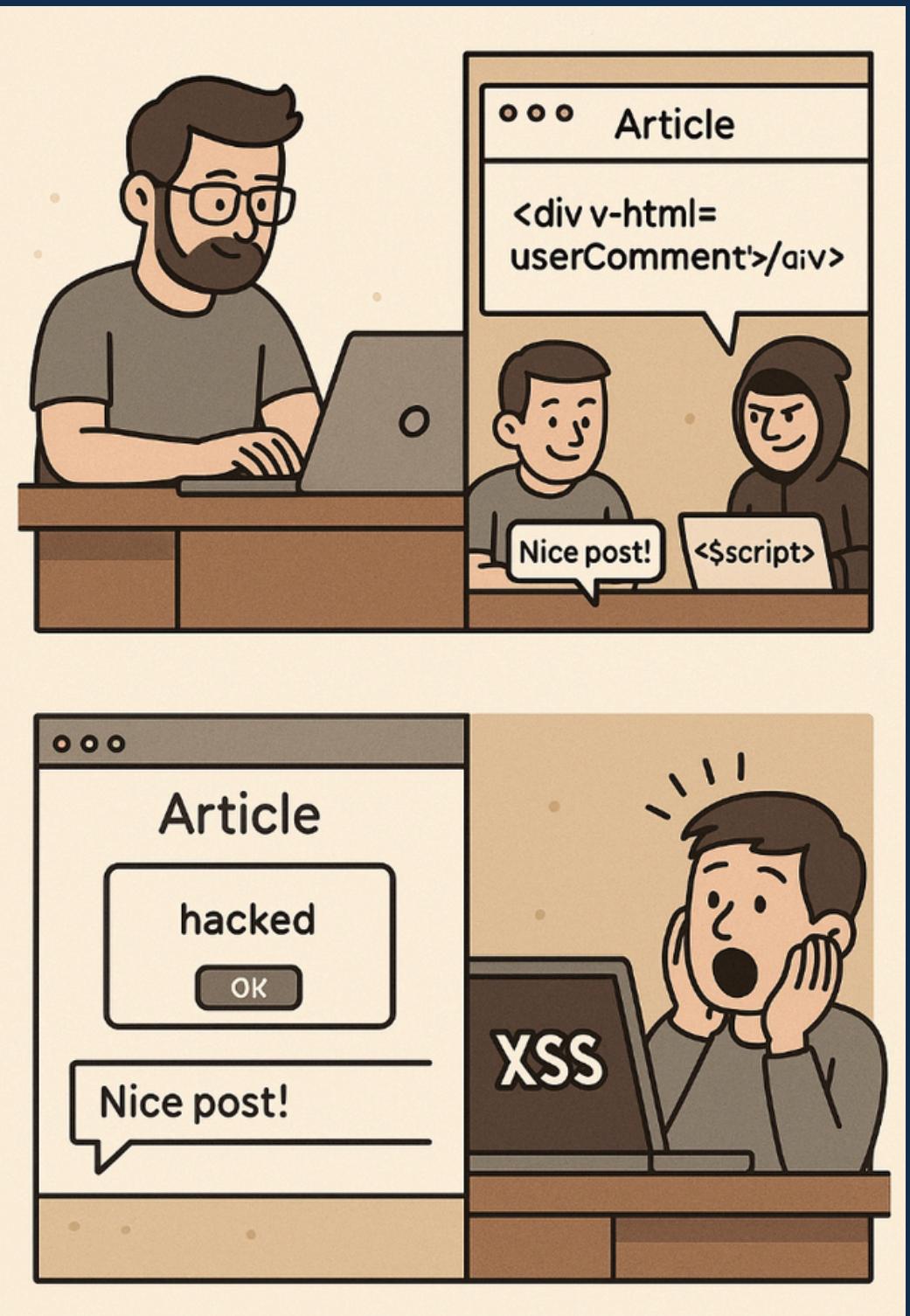
QUIZ

Kenapa menyimpan token di localStorage bisa berbahaya?

- A. Karena localStorage lambat
- B. Karena token bisa diakses script (XSS)
- C. Karena token tidak bisa digunakan untuk login
- D. Karena localStorage akan segera deprecated

Vue.js v-html Misuse

Kamu mau bikin halaman artikel biar support HTML dinamis, jadi pakai v-html. Eh, ternyata ada attacker yang kirim <script>alert('hacked')</script> di komentar. Halamanmu jadi ladang XSS.

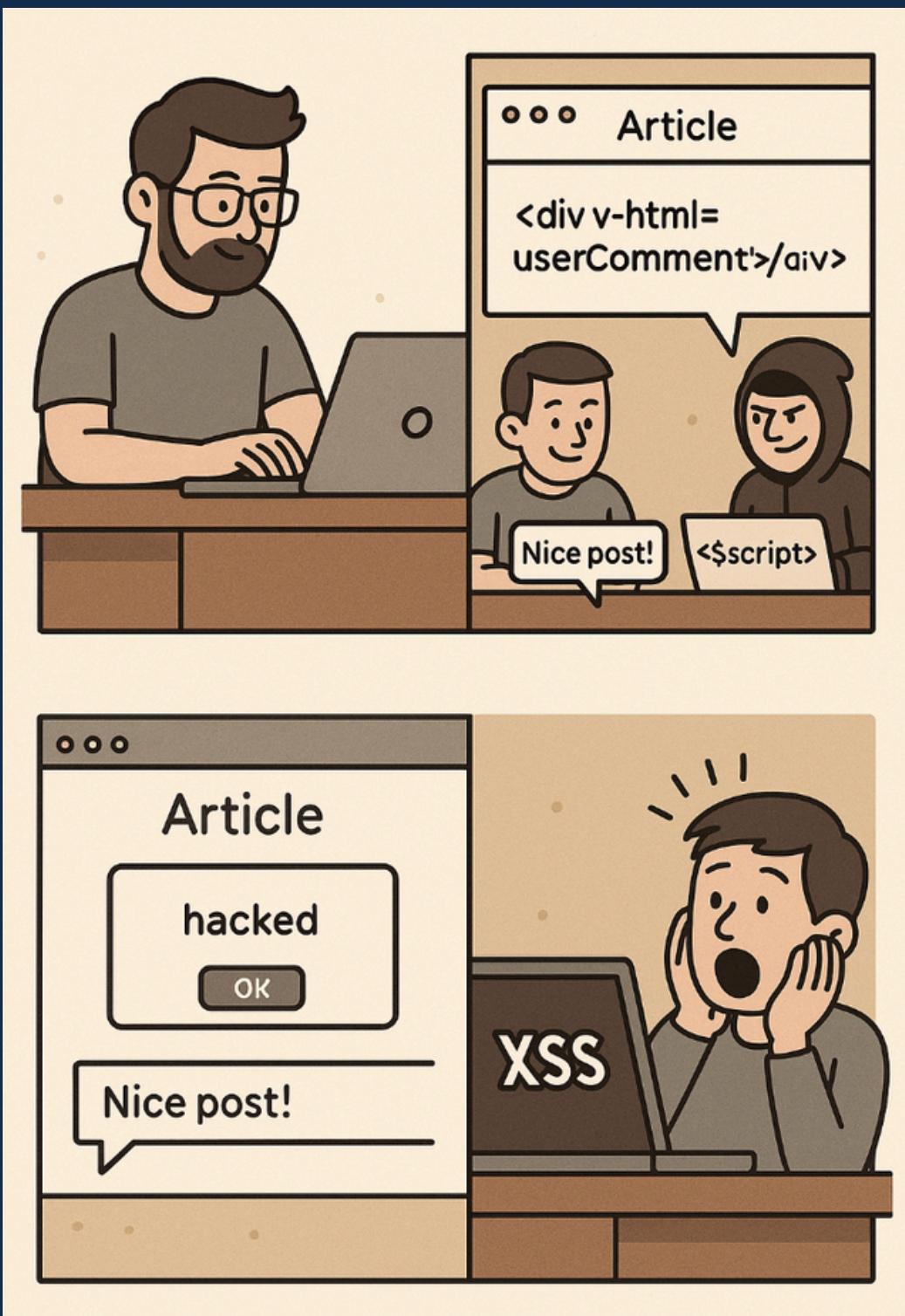


Menggunakan v-html untuk render input user tanpa sanitasi.

Root Cause

Impact

- XSS berjalan walaupun framework default aman.
- Bisa dipakai untuk mencuri cookie, session, atau deface halaman.



Before Patch

```
<div v-html="userComment"></div>
```

After Patch

```
import DOMPurify from 'dompurify';

<div v-html="DOMPurify.sanitize(userComment)"></div>
```

Best Practices

- Hindari penggunaan v-html untuk input user
- Gunakan DOMPurify atau library sanitasi
- Aktifkan Content Security Policy (CSP)
- Tambahkan linter rule untuk deteksi penggunaan v-html sembarangan
- Tambahkan unit test untuk XSS

QUIZ

Kenapa v-html bisa berbahaya jika digunakan sembarangan?

- A. Karena v-html membuat halaman lebih lambat
- B. Karena v-html bisa menjalankan HTML/JS dari input user
- C. Karena v-html menghapus semua style halaman
- D. Karena v-html hanya bekerja di browser lama

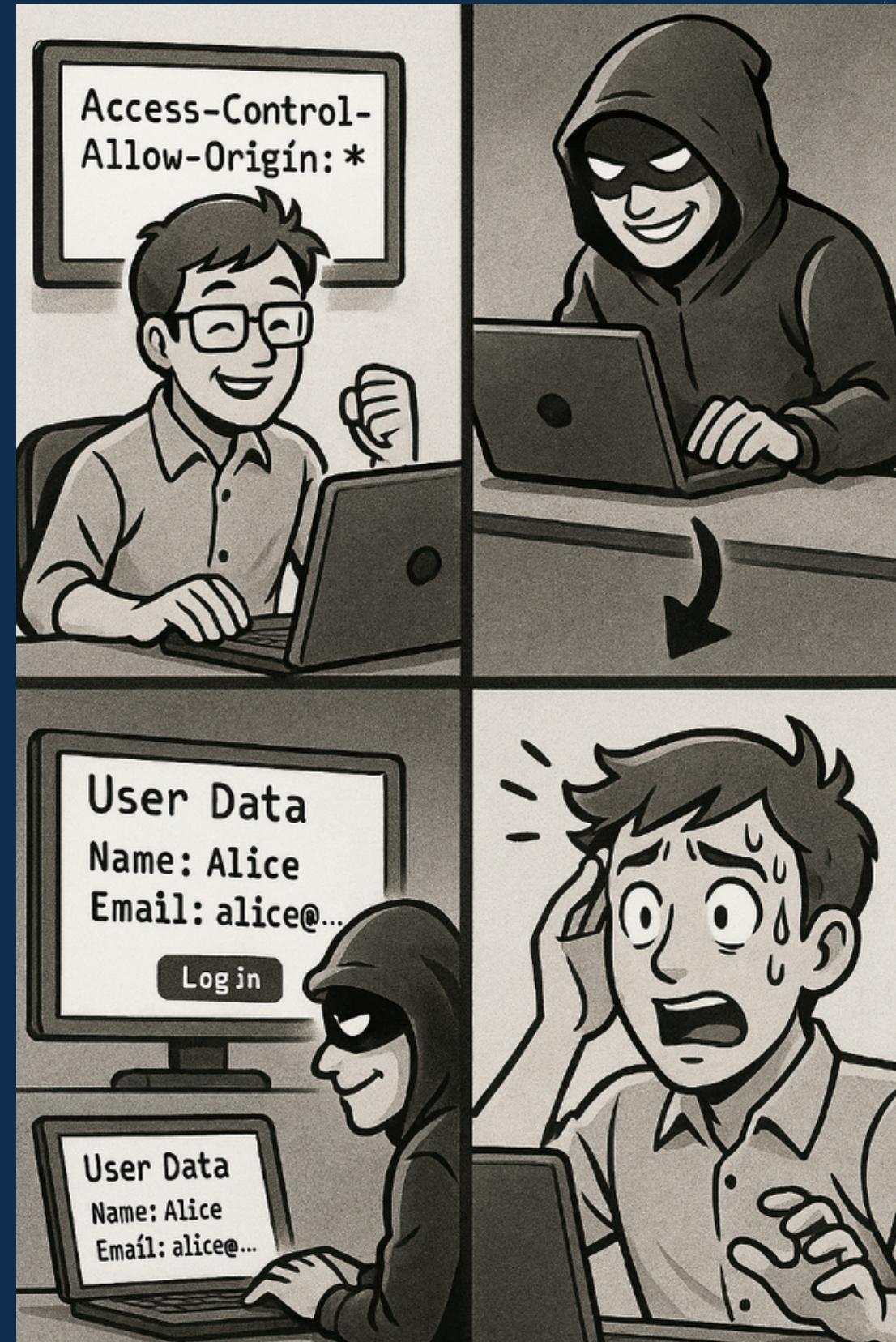


Over-Permissive CORS

Kamu set Access-Control-Allow-Origin: * biar semua API bisa dipanggil.

Eh, ternyata domain random bisa akses data user kamu langsung dari browser korban.

'Kan cuma buat testing, nanti dibenerin kok...'

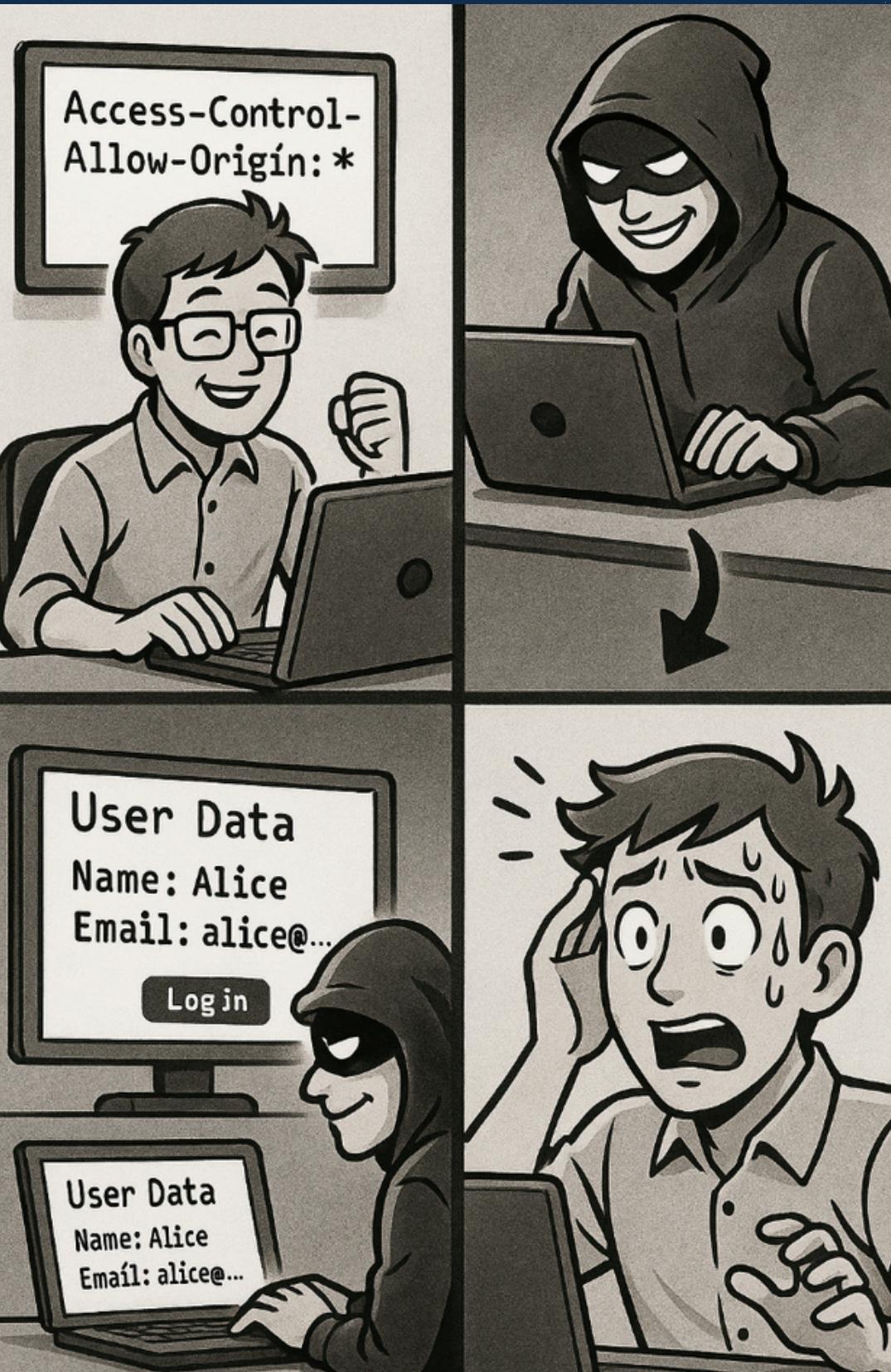


- Mengizinkan semua origin tanpa pembatasan atau validasi.

Root Cause

Impact

- Data bisa diakses website jahat lewat browser korban.
- Potensi pencurian data dan abuse API.



Before Patch

```
# Response Header  
Access-Control-Allow-Origin: *  
Access-Control-Allow-Credentials: true
```

After Patch

```
# Hanya izinkan domain resmi  
Access-Control-Allow-Origin: https://trusted.example.com  
Access-Control-Allow-Credentials: true
```

Best Practices

- Jangan gunakan Access-Control-Allow-Origin: * di production
- Gunakan whitelist origin untuk domain yang dipercaya
- Pisahkan config CORS untuk dev dan production
- Audit API secara berkala

QUIZ

Kenapa Access-Control-Allow-Origin: * berbahaya di production?

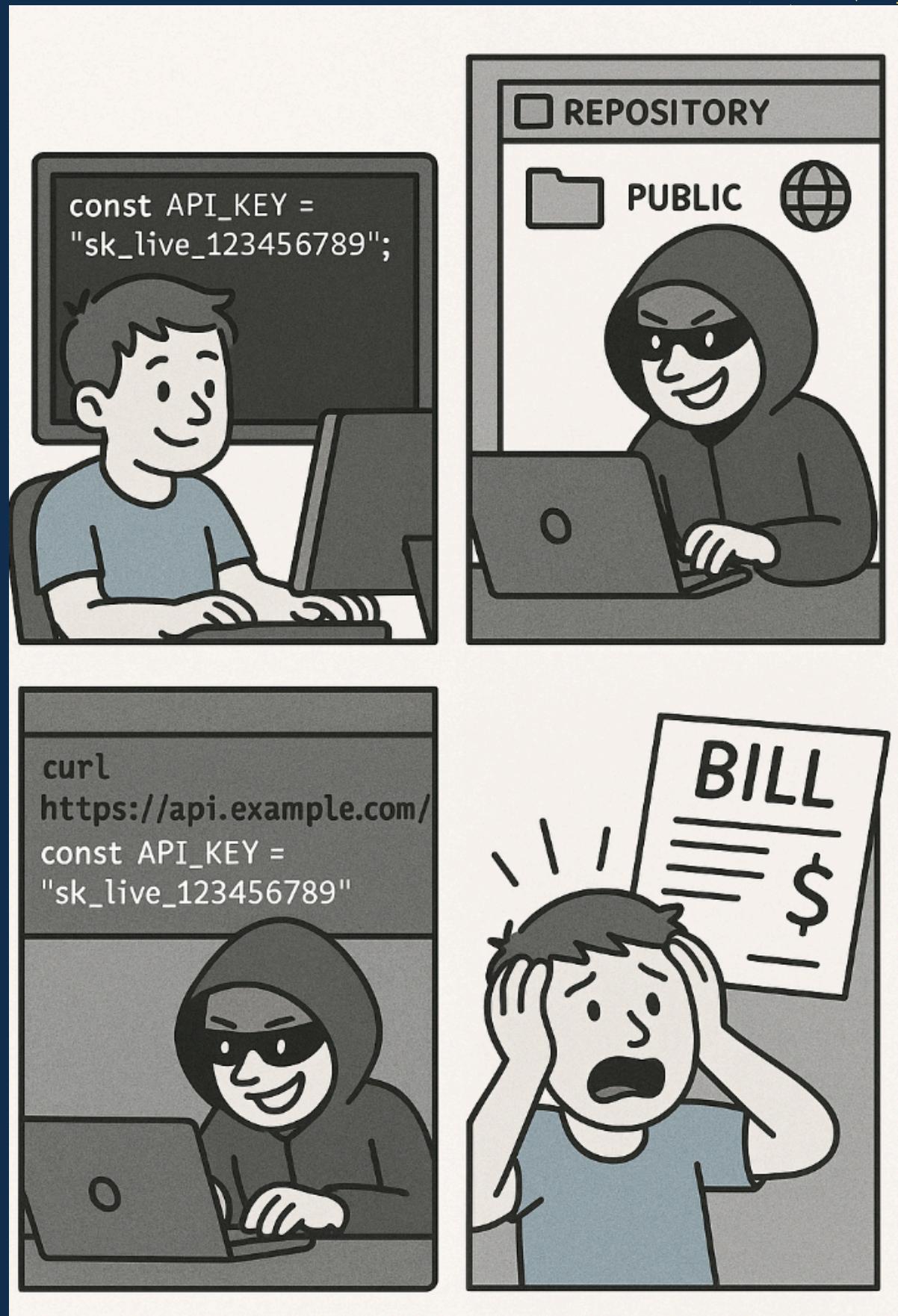
- A. Membuat API lebih lambat
- B. Membuka akses data ke website jahat
- C. Menghapus kebutuhan autentikasi
- D. Membatasi cookie



Hardcoded Secrets di File .env dan config.js

Kamu simpan API key langsung di config.js biar gampang dipakai.

Repo kamu ternyata public, dan orang lain langsung pakai API itu buat spam.

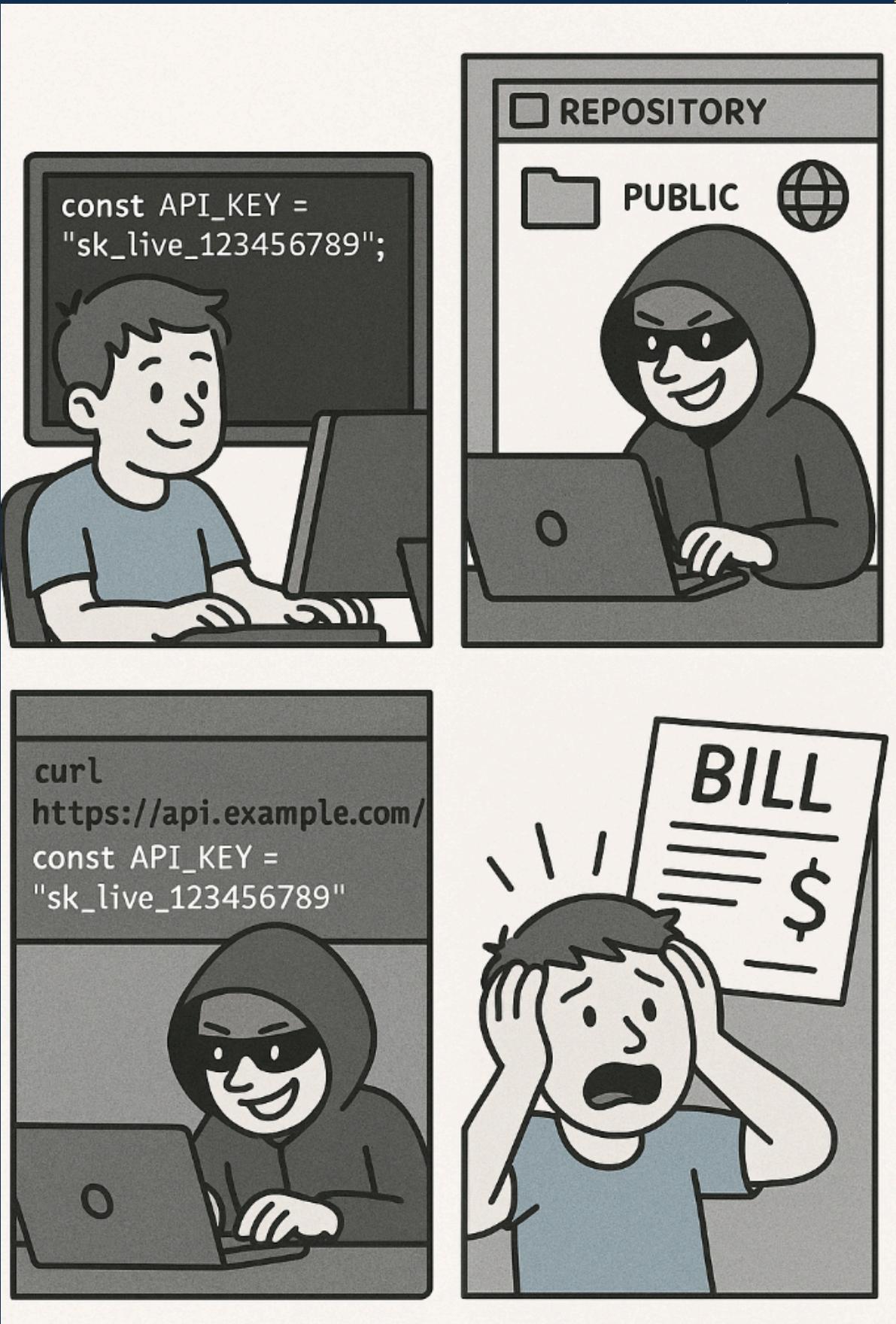


Root Cause

Menyimpan secret (API key, credential) langsung di source code yang bisa diakses banyak pihak.

Impact

Secret bocor ke public, potensi abuse API, akses sistem internal, atau pencurian data.



Before Patch

```
const API_KEY = "sk_live_123456789";
fetch(`https://api.example.com/data?key=${API_KEY}`);
```

After Patch

```
// Gunakan environment variable
const API_KEY = process.env.API_KEY;
fetch(`https://api.example.com/data?key=${API_KEY}`);
```

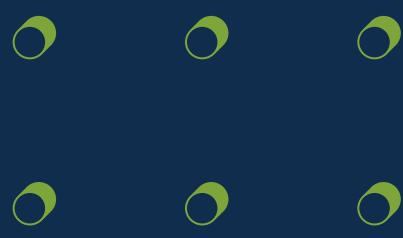
Best Practices

- Jangan hardcode API key atau credential di source code
- Gunakan environment variable untuk secret
- Gunakan vault/secret manager (misalnya AWS Secret Manager, Hashicorp Vault)
- Scan repo dengan secret scanner (GitLeaks, TruffleHog)

QUIZ

Kenapa menyimpan secret di source code itu berbahaya?

- A. Karena source code bisa diakses dan dibaca orang lain
- B. Karena secret akan hilang saat build
- C. Karena menyebabkan aplikasi lebih lambat
- D. Karena secret hanya berlaku di production



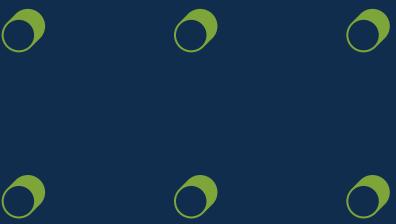
Improper Input Validation (Form Binding)

• • •
• • •

Kamu bikin form checkout,
langsung binding semua field
ke model.

Attacker ubah harga jadi
negatif dan kirim ke server.

Transaksi malah sukses,
saldo perusahaan minus.



Root Cause

Field input tidak divalidasi format dan range-nya dengan benar, atau hanya mengandalkan validasi frontend.

Impact

- Data tidak sesuai format → error atau exploit logic.
- Bisa jadi pintu masuk injeksi atau fraud.

Best Practices

- Jangan hanya mengandalkan validasi frontend
- Gunakan library validasi (Vuelidate, VeeValidate, Yup)
- Validasi format dan range di backend
- Tambahkan sanitasi input (hindari karakter berbahaya)
- Tambahkan test untuk input edge-case

QUIZ

Kenapa validasi harus ada di backend meskipun sudah ada di frontend?

- A. Karena backend lebih cepat
- B. Karena attacker bisa bypass validasi frontend
- C. Karena frontend tidak bisa memproses data
- D. Karena frontend selalu lambat



Insecure Third-Party Component

Kamu pakai library random
dari GitHub untuk fitur date
picker, biar cepat.

Minggu depan ada laporan:
library itu punya vulnerability
yang bisa eksekusi kode
berbahaya.

'Kan udah populer, pasti
aman...'



Root Cause

- Menggunakan library yang sudah deprecated atau tidak termaintain.
- Tidak pernah audit dependency atau update.



Impact

- Vulnerability di library → serangan XSS, RCE, atau data leak.
- Memperluas permukaan serangan tanpa disadari.



Best Practices

- Gunakan library populer dengan track record baik
- Audit dependency rutin (npm audit, yarn audit)
- Hapus dependency yang tidak dipakai
- Gunakan tool monitoring (Dependabot, Snyk)

QUIZ

Kenapa penting audit third-party library secara berkala?

- A. Supaya app lebih cepat
- B. Supaya dependency tidak menumpuk
- C. Supaya tahu jika ada vulnerability yang mempengaruhi app
- D. Supaya build tidak gagal

Insecure JWT Handling

- ◦ ◦
- ◦ ◦

Kamu simpan JWT token di localStorage biar gampang dipakai di frontend.

Ada bug XSS kecil, token langsung dicuri dan dipakai attacker login pakai akun user.

'Kan cuma di browser, siapa yang mau lihat?'



Root Cause

- Menyimpan token di storage yang bisa diakses JavaScript (`localStorage` / `sessionStorage`).
- XSS = semua token langsung bocor.

Impact

- Akun user diambil alih, data bocor, reputasi rusak.
- Token tidak aman untuk aplikasi dengan data sensitif.



Before Patch

```
// Simpan token di localStorage
localStorage.setItem("accessToken", token);

// Ambil token untuk request
const token = localStorage.getItem("accessToken");
```

After Patch

```
// Simpan token di HttpOnly Secure Cookie
res.cookie("accessToken", token, {
  httpOnly: true,
  secure: true,
  sameSite: "Strict"
});
```

Best Practices

- Jangan simpan token di localStorage/sessionStorage
- Gunakan HttpOnly Secure Cookie untuk token
- Hapus data sensitif saat logout
- Gunakan short-lived token + refresh token berbasis cookie
- Tambahkan validasi expiry & rotasi refresh token (backend)

QUIZ

Kenapa token jangan disimpan di localStorage?

- A. Karena localStorage lambat
- B. Karena token bisa diakses script (XSS)
- C. Karena token tidak bisa dipakai login
- D. Karena localStorage akan deprecated



Improper Error Handling (Bocorin Stack Trace)

Aplikasi kamu error di production.

User lihat layar penuh dengan error detail: file path, library, sampai query database.
Attacker yang lihat itu senyum lebar.



Menampilkan error detail (stack trace, query, file path) ke user.

Root Cause

Impact

- Attacker bisa tahu struktur internal, nama file, bahkan library yang dipakai dapat mempermudah serangan.
- User experience jelek, kelihatan “tidak profesional”.



Best Practices

- Tampilkan pesan error generik ke user
- Log detail error hanya di server (Sentry, ELK, dsb.)
- Matikan debug mode di production
- Uji error handling di staging sebelum rilis

QUIZ

Kenapa stack trace jangan ditampilkan ke user di production?

- A. Karena membuat halaman lebih lambat
- B. Karena bisa memberi informasi internal ke attacker
- C. Karena user tidak suka warna merah
- D. Karena akan menghapus semua log



Lack of Rate Limiting

User klik tombol bayar dua kali karena agak nge-lag.
Server terima dua request dan kirim dua invoice.



Root Cause

- Tombol atau form bisa dikirim berkali-kali tanpa debounce/throttle.
- Tidak ada state yang mengunci tombol selama request berlangsung.

Impact

- Double order atau double transaksi (accidental spam).
- Beban server meningkat walaupun tidak ada serangan langsung.
- Catatan: Ini bukan solusi keamanan, hanya UX improvement.



Before Patch

```
<button @click="submit()">Bayar</button>
```

After Patch

```
<button :disabled="loading" @click="submit()">Bayar</button>

<script>
export default {
  data() { return { loading: false } },
  methods: {
    async submit() {
      this.loading = true
      await apiPay()
      this.loading = false
    }
  }
}
</script>
```

Best Practices

- Disable tombol saat request berlangsung
- Gunakan debounce/throttle untuk input yang sering berubah (misal search bar)
- Tampilkan loading state agar user tahu request sedang berjalan
- Ingatkan bahwa rate limiting sebenarnya harus di backend untuk keamanan

QUIZ

Kenapa frontend rate limiting tidak cukup untuk keamanan?

- A. Karena attacker bisa langsung akses API tanpa UI
- B. Karena frontend lambat
- C. Karena rate limiting frontend memperlambat user
- D. Karena frontend tidak bisa memunculkan loading state



QnA



Common Threat & Best Practices : VueJS Frontend Development

Common Issues Repo

QnA

94





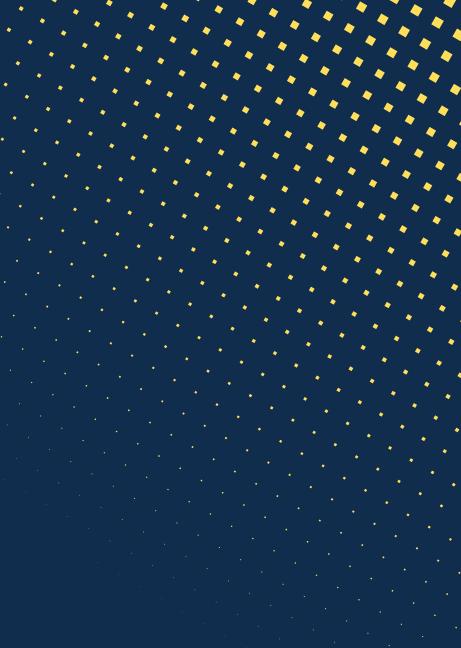
Hands-On : Temukan Security Bug di Aplikasi Kedatangan Wisatawan Luar Negeri



REPO URL



API DOC URL



Presentasi :

Temukan Security Bug di Aplikasi Kedatangan Wisatawan Luar Negeri



QnA

98

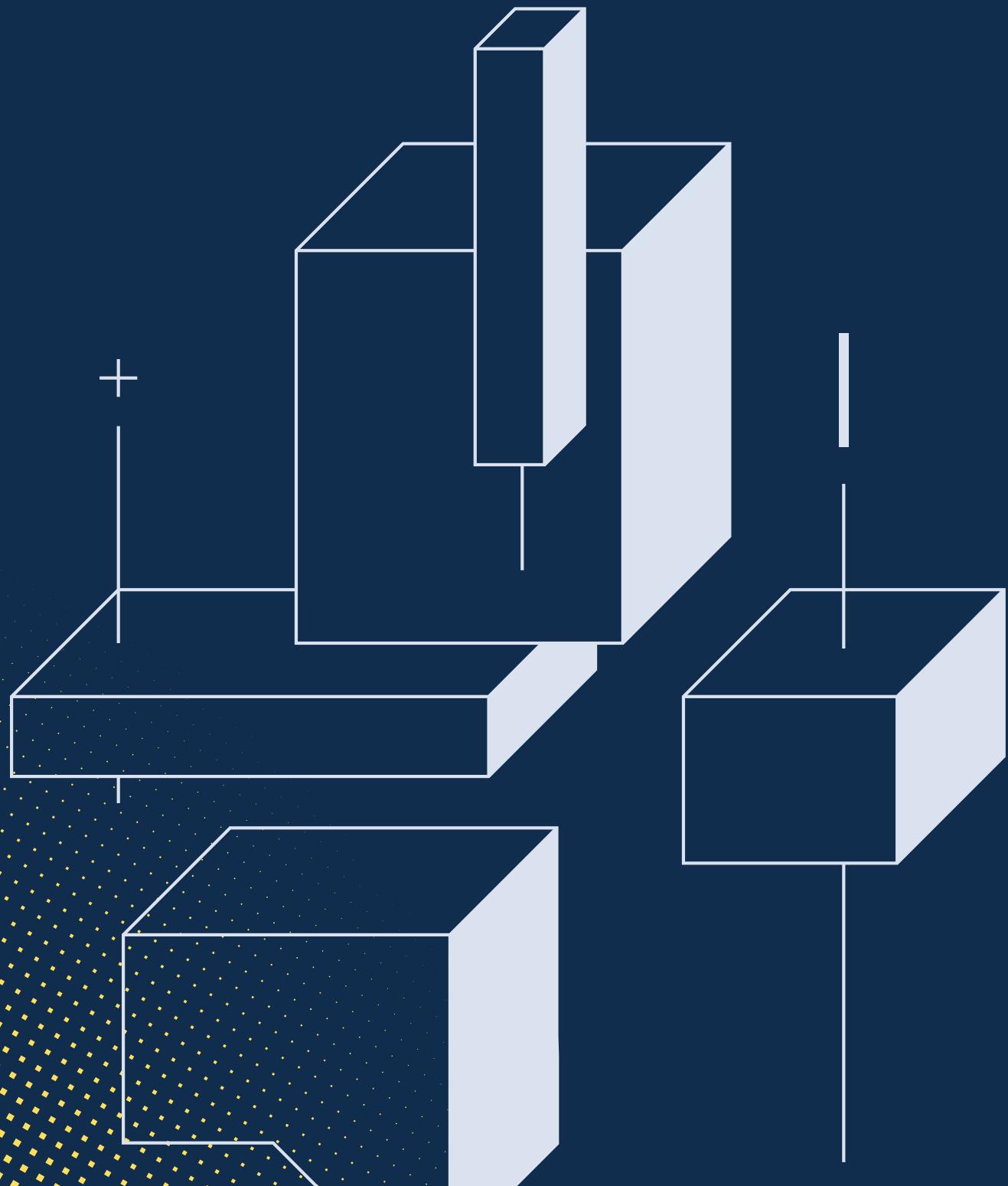


PATCH THE APP



QnA

100



Terima Kasih



HARS CODE

www.harscode.dev