📋 Risk Assessment Table

Risk ID	🧨 Risk / Threat	🔁 Trigger / Cause	🎲 Likelihood	💥 Impact	🛡️ Contingency Plan
1	Cross-Site Scripting (XSS)	Unsanitized query parameter `tab` in the URL allows script injection	Medium – users can freely manipulate the URL	High – may cause script injection, phishing redirects, or UI failure	Validate and sanitize query parameters. Default to safe values for invalid inputs. Avoid direct user input injection in rendering logic.
2	Insecure HTTP Links	Accidentally using `http://` instead of `https://` in source code	Low – mostly occurs during development. Nowadays, the browser security policies typically catch such errors early	Medium – risks include MITM attacks and browser security warnings	Enforce HTTPS usage via ESLint rule that flags `http://`. Always use `https://` for all external resources to ensure secure communication.
3	Exposed Cloudinary URLs	Predictable or permanent URLs used for sensitive image assets	Medium – private files are accessible by URL in our code	High – sensitive content may be leaked or scraped	Use signed, expiring Cloudinary URLs. Implement authenticated delivery. Future work required to configure Cloudinary-level access control.

🧠 Explanation

Risk 1: XSS via `tab` Query Parameter

The website uses the query string `?tab=videos` (or `chapters`, `quizzes`) to determine the active tab in the interface, for example: `https://iconceptsorthodontics.vercel.app/?tab=videos` will lead the user to the video tab.

However, if `tab` is not one of the expected strings, let's say the user modifies the `tab` value to an unexpected string such as: `?tab=<script>alert("XSS")</script>` the page either breaks (e.g., shows a blank screen) or becomes vulnerable to a Cross-Site Scripting attack since the inputs were not properly sanitized.

🛡️ Data Protection & Secure Coding Practices

This vulnerability was identified during unit testing, where we simulated malicious and invalid inputs to the `tab` parameter. As a secure coding response:

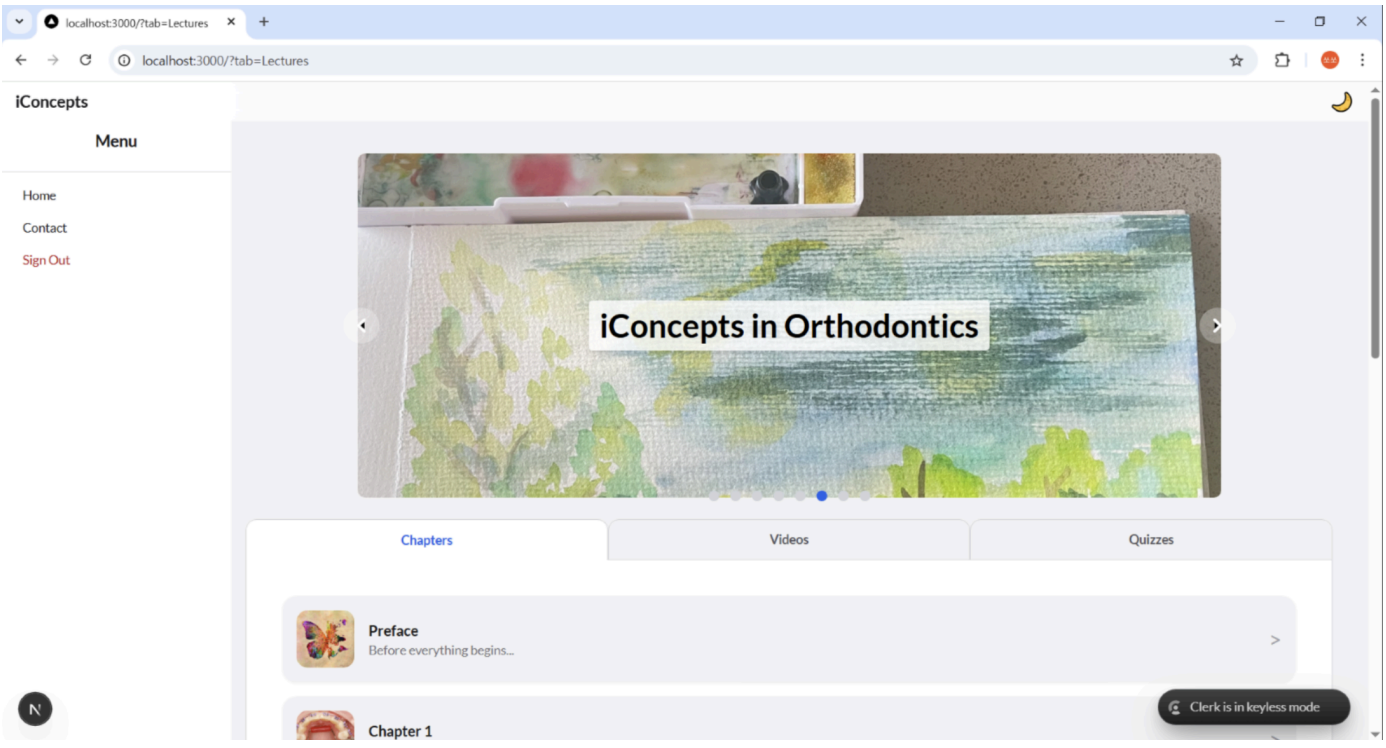
- Explicitly validate all user-controlled inputs, especially those that influence UI state or affect DOM logic.
- Avoid using raw query parameters directly and instead define a whitelist of valid options (`'chapters'` , `'videos'` , `'quizzes'`).
- Enforce strict typing with as const and typeof validTabs[number] to prevent coercion or injection.
- Apply defensive defaults (i.e., fallback to 'chapters') for invalid or missing values.

⚠️ Original Code (Insecure)

```
const initialTab = (searchParams.get('tab') as 'chapters' | 'videos' | 'quizzes') || 'chapters';
const [activeTab, setActiveTab] = useState<'chapters' | 'videos' | 'quizzes'>(initialTab);
```

✅ Improved Code - Safe Fallback and Type-Safe

```
const validTabs = ['chapters', 'videos', 'quizzes'] as const;
type TabType = typeof validTabs[number];

const tabParam = searchParams.get('tab');
const initialTab: TabType = validTabs.includes(tabParam as TabType)
    ? (tabParam as TabType)
    : 'chapters';
```



🧪 Security Integrated in Development Workflow

- **Unit tests** were written to verify fallback behavior and input validation, using examples such as `?tab=` , `?tab=random` , and `<script>` payloads.

- These tests were run both locally and on the deployed site to confirm consistent handling of malicious input.
- This change is now part of our core React component logic, integrated at component load time, ensuring defense-in-depth even for direct URL access.

Risk 2: Insecure `http://` Links in Codebase

Accidentally referencing external resources over `http://` instead of `https://` can lead to several security issues, including:

- **Man-in-the-Middle (MITM)** attacks where attackers intercept or modify unencrypted requests.
- **Mixed content warnings** in browsers, causing user distrust.
- **Blocked resources**, such as scripts or images, especially in modern secure-by-default browsers.

This risk was identified during a code review when a third-party image URL was inadvertently included using the `http://` protocol.

🛡️ Data Protection & Secure Coding Practices

To eliminate the risk of insecure URLs being introduced into the codebase, the following secure coding measures were taken:

- Implemented a custom ESLint rule to flag any usage of `http://` within the code.
- Enforced this rule across all relevant files (`.js` , `.ts` , `.tsx`) as part of the linting process.
- Integrated this check into the development lifecycle to ensure early detection during development, not after deployment.

⚠️ Original Risk (Unrestricted Use)

Previously, developers could unknowingly introduce insecure resources:

```
<img src="http://example.com/logo.png" />
```

This could silently pass through code reviews and CI if not manually detected, leading to mixed content or MITM vulnerabilities in production.

✅ Improved Code – Mitigation with ESLint Rule

We implemented the following ESLint rule to **warn developers** any time a literal value begins with `http://` :

```
"no-restricted-syntax": [
  "warn",
  {
    selector: "Literal[value^='http://']",
    message: "Use HTTPS instead of HTTP for external resources.",
  },
```

```
  ],
```

Full Integration Example

```
import { dirname } from "path";
import { fileURLToPath } from "url";
import { FlatCompat } from "@eslint/eslintrc";

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

const compat = new FlatCompat({
    baseDirectory: __dirname,
});

const eslintConfig = [
    ...compat.extends("next/core-web-vitals", "next/typescript"),
    {
        files: ["**/*.js", "**/*.ts", "**/*.tsx"],
        rules: {
            "no-restricted-syntax": [
                "warn",
                {
                    selector: "Literal[value^='http://']",
                    message: "Use HTTPS instead of HTTP for external resources.",
                },
            ],
        },
    },
];

export default eslintConfig;
```

🧪 Security Integrated in Development Workflow

- The ESLint rule is part of the core project configuration, ensuring checks run during each `lint` command and CI/CD pipeline.
- Developers receive real-time warnings in their IDE or terminal if they use `http://` URLs.
- This proactive safeguard enforces HTTPS usage consistently across the codebase and prevents insecure practices from reaching production.
- Unit test conducted on both localhost and deployed website, including changing the beginning of some media URLs to `http://` randomly.

Risk 3: Exposed Cloudinary CDN URLs

Cloudinary provides a public CDN (`https://res.cloudinary.com/...`) to deliver uploaded assets. However, **directly embedding static links to private or sensitive assets** (e.g., profile photos, academic content, health media, internal training videos) introduces the following risks:

- **Unauthorized access** by anyone who guesses or intercepts the URL.

- **No access control**, rate-limiting, or expiration.
- **Untraceable exposure** when URLs are shared outside intended boundaries.

Example of insecure usage:

```
<img src="https://res.cloudinary.com/demo/image/upload/v1234567890/private-image.jpg" />
```

Even if the intent was privacy, anyone with this link can access the resource without restriction.

🛡️ Data Protection & Secure Coding Practices

To mitigate this risk, the app now uses **signed Cloudinary URLs** that:

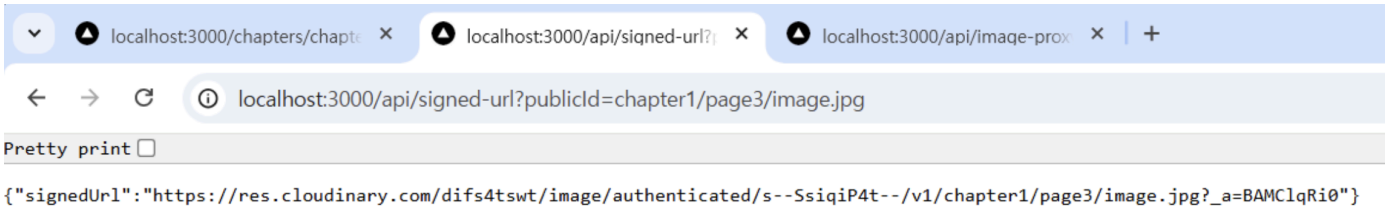
- Require a secret signature to be valid.
- Include an expiration time (e.g., 1 hour).
- Are only generated on-demand via a backend API, preventing front-end tampering or URL sharing.

This ensures secure, time-limited access and aligns with **privacy-by-design** principles.

✅ Improved Code - Dynamic Signed URL API

```
// pages/api/signed-url.ts
import type { NextApiRequest, NextApiResponse } from 'next';
import { v2 as cloudinary } from 'cloudinary';

cloudinary.config({
    cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
    api_key: process.env.CLOUDINARY_API_KEY,
    api_secret: process.env.CLOUDINARY_API_SECRET,
});

export default function handler(req: NextApiRequest, res: NextApiResponse) {
    const publicId = req.query.publicId as string;

    if (!publicId) {
        return res.status(400).json({ error: 'Missing publicId' });
    }

    const signedUrl = cloudinary.url(publicId, {
        type: 'authenticated',
        sign_url: true,
        expires_at: Math.floor(Date.now() / 1000) + 600, // 10 minutes
    });

    return res.status(200).json({ signedUrl });
}
```

This API dynamically generates signed URLs like the image below, limiting access time and enforcing authenticated delivery — significantly increasing security.



❗ Remaining Gap

While the signed URL API now supports secure, time-limited delivery, the **fix is not fully complete** because:

- Previously uploaded assets remain under public delivery mode (`type: 'upload'`), which means they can still be accessed via static URLs.
- Cloudinary does not support bulk conversion** of existing assets to `authenticated` , and many static links are already embedded across content and components.
- A manual refactor of all references would require significant QA and regression testing effort, which is not feasible within the current sprint timeline.

As a result, we **retain the current public URLs** for backward compatibility while preparing infrastructure for a future transition to authenticated delivery.

🔁 Future Action

To close this gap and harden content delivery in upcoming development cycles:

- Document the migration path for converting existing assets to `type: 'authenticated'` , including update scripts and refactoring points.
- Mandate secure asset delivery: All new uploads should explicitly set `{ type: 'authenticated' }` via backend or admin tooling.
- Define developer onboarding docs that emphasize security-conscious third-party integrations, including:
  - Valid usage patterns for Cloudinary signed URLs.
  - Avoidance of hardcoding static URL.
  - Guidance on expiring delivery links and secure upload workflows.
- Ensure CI/CD pipelines and code reviews in the future include checks for `res.cloudinary.com/.../upload` in source code to catch regressions.