

脆弱性診断結果（対象：ASP.NET WebForms / C# コード DocumentAsk）

総評

- 大きな構造上の問題（SQL インジェクション等の直撃）は現時点で見当たりません。DB クエリはパラメータ化されており、表示時のエンコードも一部は適切です。
- 一方で、以下の高リスク項目が残っています。
 - 例外メッセージのそのまま表示による反射型 XSS および情報漏えい
 - CSRF 対策の欠如（外部 API 呼び出し・費用発生の観点で重大）
 - 外部 AI へ社内ドキュメントを送信するデータ持ち出しリスク（ガバナンス/規約順守）
 - DoS/コスト増大リスク（全文読み込みの RAG・長文入力無制限・再試行/外部 API 連打）
 - API キーの保管方法（平文 appSettings）
 - 古い iTextSharp 利用（EOL による既知脆弱性の懸念）

以下、詳細と対策案です。

高リスク

1) 例外メッセージの直出し（反射型 XSS/情報漏えい）

- 問題:

- lblMessage.Text = "エラー:" + ex.Message; のように例外メッセージをそのまま出力しています。

- Label は HTML エンコードされずに出力されるため、外部サービスのエラーメッセージや攻撃者が混入させた文字列により XSS が成立します。

- また「Chat API エラー: {StatusCode} / {json}」「Embedding API エラー: ... / {json}」とサーバー内部情報を露出しています。

- 影響:

- XSS、内部構成やスタックトレースの漏えい、運用情報の露出。

- 対策:

- ユーザー表示用は必ず HTML エンコードし、詳細はログのみに出す。

- 例:

```
try { ... }
```

```
catch (Exception ex)
```

```
{
```

```
    lblMessage.Text = HttpUtility.HtmlEncode("エラーが発生しました。管理者にご連絡ください。");
```

```
    DBLogger.LogError("DocumentAsk.btnAsk_Click", ex); // 詳細はログのみ
```

```
}
```

- すべての lblMessage.Text 代入箇所と同様に適用。

2) CSRF 対策の欠如

- 問題:

- ボタンの POST で外部 API（OpenAI）を叩き、コストが発生し得ますが、CSRF 対策が見えません。

- 影響:

- 攻撃者が被害者のブラウザから勝手に問い合わせを発生させ、API 料金・リソース消費を誘発。

- 対策（WebForms 推奨パターン）:

- ViewStateUserKey の設定（ユーザー/セッション固有化）

```
protected void Page_Init(object sender, EventArgs e)

{

    this.ViewStateUserKey = Session.SessionID; // or User.Identity.Name

}
```

- EventValidation/ValidateRequest を有効化（web.config / .aspx）

- 可能ならば独自の Anti-CSRF トークン（HiddenField+Cookie のダブルサブミット）を導入し、PostBack 時検証。

- 重要操作（外部 API・ファイル出力）には追加で ReCAPTCHA 等の導入も検討。

3) データ持ち出し（外部 AI への社内情報送信）

- 問題:

- RAG により DB から抽出した社内ドキュメント抜粋を、そのまま OpenAI API へ送信。

- 影響:

- 機密情報・個人情報等の第三者提供。委託/規約・契約/GDPR 等への抵触リスク。

- 対策:

- 利用規約・社内規程の整備と同意フロー。

- 機密区分のフィルタリング/マスキング（個人情報、秘匿語の匿名化）。

- モデル/リージョン選択の見直し（企業向けプライバシープラン等）。
- 監査ログ（質問/抜粋/出力の追跡）を安全に保管。

4) DoS/コスト増大リスク

- 問題:

- RAG で DocumentEmbeddings 全件を読み込み（タイトルフィルタなし時）。件数が多い場合、CPU/メモリ/DB IO を圧迫。

- 質問文や RAG コンテキストの長さ制限なし。外部 API のトークン/コストを無制限に消費しうる。

- PostWithRetry の再試行+長いタイムアウト（2分）で同時多重実行時にスローダウン/枯渇。

- 影響:

- アプリの応答遅延・スレッド枯渇・コスト爆発。

- 対策:

- 入力長のハードリミット（例：最大文字数や推定トークン数）。超過時はエラー返却。

- RAG の DB アクセスを制限

- 件数上限（SELECT TOP N）、ベクトル検索を DB 側で実施（pgvector/SQL Server のベクトル機能等）。

- CommandTimeout の明示（短めに設定）。

- 外部 API 呼び出しのレート制限（ユーザー/IP/セッション単位）。

- PostWithRetry の最大再試行・バックオフの上限明確化、キャンセルサポート（CancellationToken）。

5) API キーの保管方法

- 問題:

- appSettings の平文 (ConfigurationManager.AppSettings["OpenAI_API_Key"])。構成ファイルの漏えい時に直撃。

- 対策:

- 機密情報はセキュアストアに保管 (例: Azure Key Vault、AWS Secrets Manager、DPAPI で web.config の appSettings セクション暗号化)。

- 実行時に参照し、ログや例外にキーを絶対に出さない。

- Authorization ヘッダ設定はアプリ起動時に一度だけ設定 (現状でも静的 HttpClient だが、Header 設定箇所はアプリ起動フローへ移動推奨)。

中リスク

6) エラーメッセージの過度な詳細表示

- 問題:

- API エラー時にサーバーが受け取った JSON をそのまま混在表示。

- 対策:

- ユーザーには「処理に失敗しました」程度。詳細はログのみ。

7) iTextSharp の EOL/既知脆弱性懸念

- 問題:

- iTextSharp（5 系）は EOL。既知脆弱性への対応が期待できません。
- 対策:
- 可能なら iText7 系（iText 7）等、メンテされているライブラリへ移行。

8) ダウンロード応答のセキュリティヘッダ

- 問題:
- ダウンロード応答にセキュリティヘッダなし。
- 対策（任意だが推奨）:
- X-Content-Type-Options: nosniff
- X-Frame-Options: DENY（全体的にも）
- Content-Security-Policy: default-src 'none'; frame-ancestors 'none'; sandbox
- Referrer-Policy: no-referrer
- HSTS（HTTPS のみ運用時、グローバルで）

9) TLS バージョン強制

- 対策（.NET Framework 利用時は特に）:
- ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12 | SecurityProtocolType.Tls13;
- サーバー側の暗号スイート/プロトコルの強化。

10) SQL/JSON 取り込みのサイズ検証

- 問題:

- Embedding 列の JSON をそのままデシリアライズ。巨大化/破損時の負荷。

- 対策:

- Embedding 列のサイズ上限バリデーション、try-catch で安全に弾く。

- 期待次元と一致しない配列は破棄。

低リスク・改善

11) 出力ファイルのキャッシュ制御

- 対策:

- `Response.Cache.SetCacheability(HttpCacheability.NoCache);`

- `Response.Cache.SetNoStore();`

12) 権限チェック

- 問題:

- このページの利用者制御がコード上からは確認できず。

- 対策:

- BasePage/認証がある前提でも、ロールに応じて RAG の対象ドキュメントを制限。

13) 例外の包括的ハンドリング

- 対策:

- CreateJsonContent/FindSimilarChunks 等、例外時に外へ生で投げず、上位で一律にユーザ向け無害化+ログ。

具体的修正例（抜粋）

A. ユーザー表示メッセージのエンコードと秘匿化

- すべての lblMessage.Text 代入を以下のように変更

```
lblMessage.Text = HttpUtility.HtmlEncode("エラーが発生しました。管理者にご連絡ください。");
```

```
DBLogger.LogError("...", ex);
```

B. CSRF 対策（最低限）

- Page_Init で

```
this.ViewStateUserKey = Session.SessionID;
```

- 可能なら独自 Anti-CSRF トークンを HiddenField+Cookie で実装し、PostBack 時に検証。

C. 入力長の制限（質問）

- 例（4,000 文字程度など、要件で調整）

```
string question = txtQuestion.Text.Trim();
```

```
if (question.Length > 4000) { ...エラー... }
```

D. RAG の DB 読み出し上限/タイムアウト

- `SqlCommand.CommandTimeout = 10;`（要調整）

- 件数上限やベクトル検索の DB 側実装（将来的にはインデックス活用）を検討。

E. API キーの秘匿化

- `web.config` の `appSettings` ではなく、保護された構成（DPAPI）や Key Vault を使用。

- 初期化時のみ `_http.DefaultRequestHeaders.Authorization` を設定。

F. セキュリティヘッダ（ダウンロード応答直前）

- 例:

```
Response.Headers["X-Content-Type-Options"] = "nosniff";
```

```
Response.Headers["X-Frame-Options"] = "DENY";
```

```
Response.Headers["Referrer-Policy"] = "no-referrer";
```

G. iTextSharp の見直し

- 可能であれば iText 7 等のサポート継続ライブラリへ移行。

H. 例外/エラー JSON のユーザー画面表示禁止

- 「Chat API エラー: ... / {json}」といった詳細はログにのみ記録。

確認事項/運用面

- 外部 AI へのデータ持ち出しの規約・契約と監査ログ設計。

- RAG の対象文書に対するアクセス制御（ユーザー権限でフィルタ）。
- レート制限/同時実行数/キュー制御（アプリ全体でのスロット管理）。

まとめ

- まずは「XSS/情報漏えい（例外の直出し）」「CSRF」「外部 AI へのデータ持ち出し方針」「DoS/コスト抑制」の 4 点を最優先で是正してください。
- その後、秘密情報の保管方法、ライブラリ更新、ヘッダ強化、入力/出力のサイズ・件数制限などのハードニングを段階的に実施することを推奨します。