ChatGPT

Redesigning the Web App Security Checklist (One-Page HTML App)

Part A – Findings & Recommendations

Current Application Overview & Data Flow

The existing **Web App Security Checklist** is a single-file HTML tool that runs entirely in-browser (no server needed) [1] [2]. It presents security test items organized as **Category → Sub-Category → Item** cards, each item with a checkbox (and optional sub-task checkboxes) and an expandable "Details" section for metadata [2] [3]. A fixed toolbar provides search filtering, an open/completed toggle, local save, CSV/JSON export, reset, and import functions [2]. User progress (checked items and edited details) is saved to `localStorage` so that reopening the file retains the state [3].

Data can be imported or exported in **CSV or JSON** formats [4] [5]. The CSV schema uses a **required header** `Category, Sub Category, Item, Done, Subtask, Subtask Done, ASVS, Description, Tools, Links, Applicability, Sources, Tags` [6]. Each CSV row represents either an item (if "Subtask" is blank) or a sub-task (if "Subtask" is filled) under the previously mentioned item [7]. The JSON import accepts either a flat array of row objects (same keys as CSV) or a nested array of category objects with subcats/items [5]. Internally, the app normalizes these inputs into a nested `data` structure: an array of category objects, each with `subcats` list, each containing `items` list, and each item having properties like `done`, `title`, `tags[]`, `subtasks[]`, and detail fields (`asvs`, `desc`, `tools`, etc.) [3]. On load, it either initializes from an embedded dataset or from an existing `localStorage` JSON string [8].

UI/UX: Categories are displayed as cards in a responsive grid. Clicking a category header toggles collapse/expand of its subcategories and items. Each sub-category section lists its items with a checkbox and title; any tags appear as clickable pills (which filter by tag) [9]. Sub-tasks (if present) are indented under their parent item [10]. The Details panel (initially collapsed) for an item shows textareas for ASVS, Description, Tools, Links, Applicability, and Sources [11]. Edits in these textareas are saved to `data` on every input event (persisting to localStorage) [12]. A search box filters items by keywords in category, subcategory, title, tags, or ASVS fields (the current implementation concatenates those into a haystack) [13]. A dropdown filter ("All/Open/Completed") is present to show only unchecked or checked items, though this filter logic appears to be incomplete in code (the `#filter` control exists but its value isn't currently applied in the render loop). The app supports marking all items in a category or sub-category as done via master checkboxes on category/subcat headers [14] [15]. Progress counts (completed/total) are shown per category and overall [16] [17].

Data Handling: Importing a CSV/JSON merges into the existing data by ensuring the category/subcategory/item hierarchy is created, then adding any new sub-tasks and merging detail fields. For example, the code `mergePrefillRows()` merges each row: it finds or creates the category, subcat, and item, sets the `done` flags (True/False), appends any subtask text and done status, and concatenates detail text into

existing fields (allowing multiple import files to build up an item's description) [18] [19]. Tags from CSV are split on whitespace/commas and merged uniquely into an item's tag list [20]. The **Export** functions output either the in-memory `data` as pretty JSON or reconstruct the CSV with the same header as import [21]. The current CSV exporter iterates through all categories, subcats, and items; if an item has sub-tasks, it emits each sub-task as a separate CSV row (repeating the item's info) [22]. This ensures the CSV can round-trip back into the app [7].

Gaps and Limitations in the Current Implementation

Overall, the app is functional and lightweight, but our analysis identified several areas for improvement:

- **No "Hide by Default" or Priority Features:** The current model has no concept of marking certain items or sub-tasks as hidden or of differing priority levels; all items display by default. The user request specifically calls for a boolean "Hide By Default" flag on items/sub-tasks and a priority level (0=very low up to 3=high) to aid focus. These are not present in the existing data schema or UI. We will need to extend the data model to include `hidden` and `priority` fields and update the UI to respect them (e.g. hide items unless a "Show hidden" toggle is active, and visually indicate priority).

- **Filtering and UI Controls:** While the UI has a search and an open/done filter, the "Open/Done filter" feature is not fully implemented in code. The `#filter` dropdown is present and triggers re-render on change [23], but the render logic does not actually omit items based on done status (no `if(f==="done")…` check in the loop). This is likely an oversight. We will implement the intended behavior: show "All" items, or only "Open" (unchecked) or only "Completed" items per the selection. Additionally, there is currently no way to filter by priority (since priority doesn't exist yet) – we'll add a priority filter control (allowing multi-select of priority levels) and also maintain the existing tag filter logic (clicking a tag pill filters by that tag [9]).

- **User Interaction for New Fields:** The current UI allows editing of detail text fields but not the item's basic properties (title, tags) via the interface – those were assumed mostly static or edited by importing data. The new **Priority** and **Hidden** attributes will also need simple editing controls. Since category/subcategory names and item titles are largely static (the README notes these key fields come from the master spreadsheet and wouldn't frequently change), we will not introduce full in-place renaming for those in the MVP. Instead, we'll provide UI elements to adjust priority and hide/show flags for each item (e.g. a dropdown to set priority and a checkbox to mark an item as hidden) in the item's Details panel. This way, users can update these properties for "existing ones" as needed [11], without cluttering the main list view. Subcategory-level **Notes** will be editable via a text area in the subcategory header area.

- **Performance Considerations:** The app renders the entire list on each relevant event (search input, filter change, checking a box, etc.). This is simple and acceptable for moderate data sizes, but if the checklist scales to hundreds of items, re-generating the whole DOM can become sluggish. Furthermore, every checkbox change calls `updateCounts()` which saves the full data to localStorage and re-renders [24] [25]. Similarly, typing in a Details textarea triggers `save(data)` on every keystroke [12]. Frequent full-data serialization could become a bottleneck as data grows (localStorage writes and large DOM updates are relatively slow). In testing small samples this is fine, but for a fully populated OWASP checklist (potentially many dozens of categories and hundreds of items), we flag this as a potential performance bottleneck. We recommend in future to consider

debouncing the save on text input (save after a short delay or on blur rather than every character) and potentially optimizing render updates (e.g. updating just the changed item's DOM rather than full re-render). For now, our redesign will stick to the simple approach for reliability, but this is noted for future enhancements.

- **Security Analysis:** The front-end runs in a local/offline context, which limits exposure, but we still apply a security mindset:

- *Content Sanitization:* All imported data is treated as text and inserted via DOM text nodes or textarea values, not as HTML, which mitigates XSS. We confirmed the code uses safe methods (e.g., using `element.append(text)` or `textarea.value = …` for user content [11]). Even tags are inserted as text inside spans [9]. This means that any HTML or script in the source data would be displayed literally or safely contained. For example, if a malicious JSON had a description with `<img src=x onerror=alert(1)>`, it would appear as string `<img…>` in the textarea, not execute. We will maintain this strict output encoding. If in the future we allow "rich text" or clickable links, we'll implement proper sanitization (e.g. using DOMPurify to scrub HTML, or only allowing a whitelisted subset of tags).
- *Links and External Content:* Currently, the app doesn't auto-render links as anchors (links are just text in a textarea). This is actually safer for now – it avoids issues like `javascript:` URLs or file links that could be abused. We will continue to treat the **Links** field as plain text (users can copy-paste into a browser if needed). If clickable links are desired later, we'll enforce safe `http/https` schemes and add `target="_blank" rel="noopener"` to any anchor tags to prevent hijacking.
- *LocalStorage trust:* Because data persists in localStorage, a user's browser will retain the checklist data between sessions. This is convenient, but users should be aware that anyone with access to their machine/browser profile could read the checklist data (as with any localStorage data). This is an acceptable trade-off for a personal/offline tool. We recommend users export and delete data if using a shared computer.

- *CSP (Content Security Policy):* The app is designed to be opened as a local file or served statically. It currently inlines all JS/CSS, which would violate a strict CSP if hosted (since it uses inline scripts/styles). If a user wants to host it on a web server with CSP, they'd need to add a nonce or hash for the script, or we could provide an alternate version with scripts in an external file. For now, in a local context this is not an issue. Our redesign continues to be a single HTML file for simplicity, but as a future improvement we note that splitting into separate files or using a CSP-compatible approach (no `eval`, no inline event attributes – which we already avoid by using `addEventListener`) will ease deployment in hardened environments.

- **Data Model Mix and Original Spreadsheet:** The source spreadsheet ("Web Application Checklist_To Transform.xlsx") contains the master list of test cases with various columns (possibly including references, how-to-test steps, etc.). The current app doesn't directly parse XLSX (the repository showed a CSV version and some JSON samples in `scratch/` indicating a transformation step). Our approach treats the CSV as the baseline import format (which is easier to handle in-browser than XLSX). The transform logic present (`transformImportedData`) suggests they mapped complex structured fields (like `how_to_test` and `references`) from a prior JSON format [26] [27] – this is specific to the initial data preparation and not needed for daily use by end users. We will focus on a clean JSON schema for the finalized data (see Part B) rather than the interim structures. To import from the Excel in the future, an additional client-side library or pre-conversion would be needed. For

now, converting the Excel to CSV externally is acceptable (the README even offers to generate a starter CSV) [28].

Recommendation – Improve vs. Rewrite: We recommend an **iterative improvement** of the existing code rather than a ground-up rewrite. The current code is compact, understandable, and mostly achieves the core requirements. We will extend this base to add the new fields and UI controls, ensuring we don't break existing features. A full rewrite with frameworks is unnecessary given the no-build/no-install goal and the relatively simple state management needed. Instead, we'll enhance the vanilla JS architecture, keeping it modular within the single file. We'll maintain the pattern of a self-contained file, augmenting the data model and UI where needed. Future modularization (breaking into multiple files or introducing a build step) can be considered if the project grows significantly, but for the current scope (an offline checklist for individual use), the one-page approach is appropriate.

Proposed UI/UX Enhancements

To incorporate **Hide By Default**, **Priority**, and **Subcategory Notes** while keeping the interface user-friendly, we propose the following changes:

- **Collapsible Hierarchy & Layout:** We will retain the card-based layout (each Category as a card) since it allows scanning multiple categories at once on wide screens. Categories remain collapsible as now. Subcategories will stay as sections within cards; we'll add an editable **"Notes" field at the subcategory level**. This will appear as a small collapsible panel or inline text area under the subcategory title. For example, a subcategory heading could show a "Notes" toggle – clicking it reveals a text area where the user can read or enter notes for that sub-section (e.g. overarching test setup instructions). This provides context when needed without cluttering the main list. (In the prototype, we implement subcategory notes as a collapsible `<details>` under each subcategory header.)

- **Priority Indicators:** Each item will have a **priority level (0,1,2,3)** stored in data. In the UI, we will visually distinguish priorities using color cues and an optional label:

- High priority (3) items will be highlighted with a conspicuous color (e.g. a red accent bar on the left of the item card) to "stand out" [29].
- Medium (2) might use orange/amber,
- Low (1) green or blue,

- Very Low (0) a faded gray – per the request, low-priority items should appear more muted. We plan to implement this by adding a colored left border on the item container (e.g., 4px solid red/orange/green/gray depending on priority). This method is lightweight and doesn't require additional elements. We will also allow filtering by priority: the toolbar will get a **"Priority filter"** section with checkboxes for High, Med, Low, Very Low. Users can tick which priority levels to show. By default, all are checked (show all items). For example, one could uncheck "Very Low" and "Low" to focus only on Medium/High items. The filter will be multi-select (unlike the status filter which is single-select) so that combinations can be viewed. This granular control helps users focus on critical tests first. The priority values themselves will be visible when editing an item (as a number or a selection in the Details panel), but in the main list we rely on color and ordering – we assume users will generally

know the meaning of the colors or can hover to see the value in a tooltip if needed. (We do include the numeric value in the dropdown to avoid ambiguity.)

- **Hide By Default Behavior:** Each item and sub-task can be marked as "hidden". In the data, `hidden=true` means "do not show this item by default." In the UI, we will *initially hide* those elements. A new **"Show hidden items"** toggle (checkbox) will be added to the toolbar to globally reveal hidden entries. When "Show hidden" is off (default), any item or sub-task flagged as hidden is completely omitted from the DOM (not just visually hidden, to avoid confusion and to minimize clutter). When toggled on, hidden items/sub-tasks will appear, but styled distinctly (e.g. semi-transparent or italic) to indicate their optional nature. This mechanism allows the checklist to include very specialized or low-priority tests that are normally out-of-sight, while still being accessible if needed. For example, an item with `hidden=true` might be a niche test case that most projects skip; it won't distract the user unless they explicitly choose to see *all* items. We implement this by simply filtering out `hidden` items in the render loop (unless the showHidden toggle is active). Hidden sub-tasks will be treated similarly – if their parent item is visible but the sub-task itself is hidden, we won't show that sub-task unless toggled. When shown, a hidden item will carry a subtle indicator (our prototype uses a lighter opacity) so the user knows it was meant to be hidden.

- **Editing Priority and Hidden Flags:** To allow users to mark items as high/low priority or hidden/unhidden on the fly, we will add controls within the item's Details expansion. Specifically, when an item's "Details" is expanded, at the top of that panel we will provide:

- a **Priority selector** (a small `<select>` dropdown with options 0–3 labeled "Very Low, Low, Medium, High") to change the item's priority, and

- a **"Hide by default"** checkbox to toggle its hidden status. These controls will update the data immediately. If the user marks an item as hidden and they have "Show hidden" off, the item will disappear from view as soon as they collapse/refresh (the prototype will actually re-render immediately on toggle for clarity). Similarly, priority changes will immediately reflect in the item's styling (e.g., border color). This inline-edit approach avoids cluttering the main list with extra buttons or icons for these fields, but still makes the functionality accessible. It aligns with the existing pattern that item details (beyond title and checkbox) are edited in the expanded panel.

- **Subcategory Notes UI:** For subcategory-level notes, we propose a small expandable section near the subcategory header. In our prototype, each subcategory header will have a "Notes" caret that can be expanded to show a textarea (similar to item Details, but just one field). The user can enter any free-text notes (which might include context like "This subcategory is not applicable if using OAuth2" or instructions like "Ensure test user accounts are set up"). This notes field is optional; if empty, it can remain hidden unless the user chooses to add a note. We considered always showing an empty text field, but that could clutter the view if most subcats have no notes. Instead, the presence of the "Notes" toggle indicates the capability. We'll also highlight if a note exists (for instance, the summary text "Notes" could be styled differently if non-empty) so the user doesn't forget there is content. These notes are stored in the data under each subcat's `notes` property and will be preserved on export. They do not come from the original spreadsheet (the spreadsheet didn't have a notes column at subcategory level), so these will be user-provided commentary.

- **Import/Export Adjustments:** We will extend the CSV and JSON formats to include the new fields:

- **CSV:** Add new columns: `Subtask Hidden`, `Priority`, `Hidden`, and `Subcategory Notes`. The "Subtask Hidden" column parallels "Subtask" and "Subtask Done", indicating if a given sub-task is hidden by default. The `Priority` column is a number 0–3, and `Hidden` is a TRUE/FALSE for the item. `Subcategory Notes` can store a note for the subcategory; since a subcategory spans multiple rows (one per item), we will include the note text on the first item's row of each subcategory (and leave it blank on subsequent items of the same subcategory). The importer will use the first non-empty note encountered for a subcategory and apply it. This approach avoids duplicating a long note on every row while still capturing it in the CSV. We will document this in the code comment/tip for clarity. The CSV exporter will similarly output the note only on the first row of a subcategory for brevity (and blank for other rows). This ensures that re-importing the CSV yields the same note (our merge logic will ignore subsequent blanks or duplicates). The included **code comment in Part C's HTML** shows the updated CSV header and handling.

- **JSON:** For the full nested JSON form, our schema will simply include the new fields (notes, priority, hidden) as described in Part B. If using the flat JSON (array of row objects) import, we support new keys `"Subtask Hidden"`, `"Priority"`, `"Hidden"`, `"Subcategory Notes"` similar to CSV. The import logic will handle them just like other fields. For example, a flat JSON row might be `{ "Category": "X", "Sub Category": "Y", ..., "Priority": 3, "Hidden": "TRUE" }`. Our code will interpret those and set the corresponding fields. (Booleans can be given as true/false or as string "TRUE"/"FALSE"; the importer normalizes strings case-insensitively [30].) We will ensure backward compatibility: if these new columns/keys are missing, items simply get default priority 2 (medium) and hidden=false.

- **Other Enhancements and Future UX Considerations:**

- We will fully implement the **Open/Done filter** now so users can hide completed items easily (this works in tandem with priority and hidden filters). The combination of filters will be applied together (e.g., one can show only open items of High priority that are not hidden).
- The tag filtering via clicking tag pills remains unchanged (filter by one tag at a time). In the future, a more advanced tag filtering UI could be added (like a multi-select for tags similar to priority), but it's beyond our current scope.
- We considered alternate layouts (such as a left sidebar for categories, with items in a right panel). For now, we conclude the current single-page list is effective for a moderate number of categories (it provides an overview and direct search across everything). If the content grows very large (say 1000+ items), a three-pane interface or pagination might be warranted. Our priority and hidden filters, however, will mitigate overload by letting users trim the view. Additionally, if performance becomes an issue with extremely large lists, implementing **virtual scrolling** (only rendering visible items) is a potential enhancement (see Part D for a lightweight library suggestion). At this stage, we stick with the straightforward full list rendering for simplicity, given most use-cases will involve a few hundred checklist items at most.

In summary, the redesign will **add** the requested features while preserving the offline, zero-build nature of the app. The UI will gain a few new controls (priority filters and show-hidden toggle in the toolbar, plus priority/hidden inputs in item details and a notes field for subcategories) but remain clean and navigable. We prioritize a secure implementation (treating all user data as text) and will test that existing import/export continues to work with the extended schema (it will, as we only add new optional fields).

Part B – Data Schema Definition

Below is the proposed **JSON Schema** (draft 2020-12) for the checklist data structure, incorporating **Category**, **Subcategory (with notes)**, **Item**, and **Sub-task** levels. This schema assumes the "full dataset" nested form. (For flat CSV/JSON row form, the fields correspond closely but all appear in each row object as outlined after.)

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "WebAppSecurityChecklist",
  "type": "array",
  "items": {
    "type": "object",
    "required": ["category", "subcats"],
    "properties": {
      "category": {
        "type": "string",
        "description": "Category name (e.g. 'Authentication', 'Configuration')"
      },
      "subcats": {
        "type": "array",
        "items": {
          "type": "object",
          "required": ["name", "items"],
          "properties": {
            "name": {
              "type": "string",
              "description": "Subcategory name (e.g. 'Password Policy')"
            },
            "notes": {
              "type": "string",
              "description": "Optional notes for this subcategory",
              "default": ""
            },
            "items": {
              "type": "array",
              "items": {
                "type": "object",
                "required": ["title"],
                "properties": {
                  "title": {"type": "string", "description":
"The checklist item description/title"},
                  "done": {"type": "boolean", "description": "Completion
status", "default": false},
                  "hidden": {"type": "boolean", "description": "Hide-by-default
flag", "default": false},
```

```
                  "priority": {
                    "type": "integer",
                    "description": "Priority level (0=very low, 3=high)",
                    "minimum": 0, "maximum": 3,
                    "default": 2
                  },
                  "tags": {
                    "type": "array",
                    "description": "Labels/tags for grouping related items",
                    "items": {"type": "string"},
                    "uniqueItems": true,
                    "default": []
                  },
                  "asvs":   {"type": "string", "description":
"ASVS reference(s) for this item", "default": ""},
                  "desc":   {"type": "string", "description": "Detailed
description of the test", "default": ""},
                  "tools":  {"type": "string", "description": "Tools or methods
to test this item", "default": ""},
                  "links":  {"type": "string", "description": "Reference links
(OWASP, docs, etc.)", "default": ""},
                  "applic": {"type": "string", "description": "Applicability or
N/A criteria", "default": ""},
                  "sources":{"type": "string", "description": "Sources for this
test (articles, etc.)", "default": ""},
                  "subtasks": {
                    "type": "array",
                    "items": {
                      "type": "object",
                      "required": ["text"],
                      "properties": {
                        "text":   {"type": "string", "description": "Sub-task
description"},
                        "done":   {"type": "boolean", "description": "Sub-task
done status", "default": false},
                        "hidden": {"type": "boolean", "description": "Sub-task
hidden flag", "default": false}
                      }
                    },
                    "default": []
                  }
                }
              }
            }
          }
        }
      }
    }
```

```
  }
}
```

Key aspects of the schema: - The top level is an **array of Category objects**. Each category has a `category` name and a list of `subcats` (sub-categories). We use arrays to preserve order as in the original spreadsheet. - Each **Subcategory object** has a `name` and an `items` list, plus an optional `notes` field for subcategory notes (string, may be empty). - Each **Item object** represents one checklist item/test. Important fields: - `title` (string): the item's name or short description. (From spreadsheet "Item" column.) - `done` (boolean): whether it's marked completed. (From "Done" column in CSV, which was True/False.) - `hidden` (boolean): our new field for Hide By Default. Default false. - `priority` (integer 0–3): new priority level, default 2 (medium). We default to 2 assuming most items are of medium priority unless specified otherwise. - `tags` (array of strings): zero or more tags for the item. (In CSV these are entered as space/comma-separated in the "Tags" column [7]; in JSON full form we store them as an array for clarity.) - Detail fields: `asvs`, `desc` (Description), `tools`, `links`, `applic` (Applicability/N/A), `sources` – all are stored as multi-line text (strings). These correspond to various columns in the spreadsheet (ASVS mapping, description of how to test, tool suggestions, etc.). We keep them as simple strings (allowing Markdown or plaintext; no HTML). Even if multiple entries (multiple tools, links, etc.) are present, they can be separated by newlines or bullet characters in the string. (Internally the app doesn't split these; it just stores whatever text the user inputs.) - `subtasks` (array): list of sub-task objects. Each sub-task has `text` (string description), `done` (boolean status), and `hidden` (boolean hide flag). Subtasks do not carry separate priority – they're assumed to share the parent item's priority context, though we do include `hidden` for subtask-level hiding.

Sample JSON Documents:

1. **Nested JSON Example (Full Dataset Form):**

Here is a truncated example of the structured JSON for two categories, illustrating use of **priority**, **hidden**, and **subcategory notes**:

```
[
  {
    "category": "Category A",
    "subcats": [
      {
        "name": "Subcat A1",
        "notes": "",
        "items": [
          {
            "title": "Visible Item",
            "done": false,
            "hidden": false,
            "priority": 3,
            "asvs": "",
            "desc": "Example high-priority test item.",
            "tools": "",
```

```
          "links": "",
          "applic": "",
          "sources": "",
          "tags": ["auth","session"],
          "subtasks": []
        }
      ]
    },
    {
      "name": "Subcat A2",
      "notes": "This subcategory has additional context or instructions.",
      "items": [
        {
          "title": "Hidden Item",
          "done": false,
          "hidden": true,
          "priority": 1,
          "asvs": "",
          "desc": "This is a low-priority item hidden by default.",
          "tools": "",
          "links": "",
          "applic": "",
          "sources": "",
          "tags": [],
          "subtasks": []
        }
      ]
    }
  ]
},
{
  "category": "Category B",
  "subcats": [
    {
      "name": "Subcat B1",
      "notes": "",
      "items": [
        {
          "title": "Item with Subtasks",
          "done": false,
          "hidden": false,
          "priority": 2,
          "asvs": "",
          "desc": "An item with a hidden subtask.",
          "tools": "",
          "links": "",
          "applic": "",
          "sources": "",
```

```
            "tags": ["files"],
            "subtasks": [
              { "text": "Visible subtask", "done": false, "hidden": false },
              { "text": "Hidden subtask",  "done": false, "hidden": true  }
            ]
          }
        ]
      }
    ]
  }
]
```

Explanation: "Visible Item" in Category A → Subcat A1 is marked high priority (`3`) and not hidden; "Hidden Item" in Subcat A2 is marked low priority (`1`) and `hidden:true`, so it will be initially suppressed. Subcat A2 also has a `notes` string. In Category B, we show an item that has two subtasks, one of which has `hidden:true`. Note how each tag is an element in the tags array in this JSON format.

1. **Flat JSON Example (Row Objects, similar to CSV):**

For CSV or flat JSON import, each row is an object with all columns. Here's how the above data might look as flat entries (JSON array of rows):

```
[
  {
    "Category": "Category A",
    "Sub Category": "Subcat A1",
    "Item": "Visible Item",
    "Done": "FALSE",
    "Subtask": "",
    "Subtask Done": "",
    "Subtask Hidden": "",
    "ASVS": "",
    "Description": "Example high-priority test item.",
    "Tools": "",
    "Links": "",
    "Applicability": "",
    "Sources": "",
    "Tags": "auth session",
    "Priority": 3,
    "Hidden": "FALSE",
    "Subcategory Notes": ""
  },
  {
    "Category": "Category A",
    "Sub Category": "Subcat A2",
    "Item": "Hidden Item",
```

```
    "Done": "FALSE",
    "Subtask": "",
    "Subtask Done": "",
    "Subtask Hidden": "",
    "ASVS": "",
    "Description": "This is a low-priority item hidden by default.",
    "Tools": "",
    "Links": "",
    "Applicability": "",
    "Sources": "",
    "Tags": "",
    "Priority": 1,
    "Hidden": "TRUE",
    "Subcategory Notes": "This subcategory has additional context or
instructions."
  },
  {
    "Category": "Category B",
    "Sub Category": "Subcat B1",
    "Item": "Item with Subtasks",
    "Done": "FALSE",
    "Subtask": "Visible subtask",
    "Subtask Done": "FALSE",
    "Subtask Hidden": "FALSE",
    "ASVS": "",
    "Description": "An item with a hidden subtask.",
    "Tools": "",
    "Links": "",
    "Applicability": "",
    "Sources": "",
    "Tags": "files",
    "Priority": 2,
    "Hidden": "FALSE",
    "Subcategory Notes": ""
  },
  {
    "Category": "Category B",
    "Sub Category": "Subcat B1",
    "Item": "Item with Subtasks",
    "Done": "FALSE",
    "Subtask": "Hidden subtask",
    "Subtask Done": "FALSE",
    "Subtask Hidden": "TRUE",
    "ASVS": "",
    "Description": "An item with a hidden subtask.",
    "Tools": "",
    "Links": "",
    "Applicability": "",
```

```
    "Sources": "",
    "Tags": "files",
    "Priority": 2,
    "Hidden": "FALSE",
    "Subcategory Notes": ""
  }
]
```

Explanation: Each object corresponds to one CSV line. The first object is the "Visible Item" (no Subtask, so subtask fields empty). The second is "Hidden Item" (also with no subtask; note `Hidden: "TRUE"` and the Subcategory Notes filled only in this first item of Subcat A2). The third and fourth rows represent the two subtasks under "Item with Subtasks": the third row has `"Subtask": "Visible subtask"` and `"Subtask Hidden": "FALSE"`, the fourth has `"Subtask": "Hidden subtask"` and `"Subtask Hidden": "TRUE"`. Both repeat the main item info (Description, etc.) and have the parent item's Priority (2) and Hidden ("FALSE" for the item itself). In this flat format, the `Tags` field is a space-separated list in a string (e.g. `"auth session"`), as the importer will split it on spaces or commas [20]. Booleans are represented as `"TRUE"` / `"FALSE"` strings (case-insensitive) for compatibility, though our JSON importer will also accept actual `true/false` JSON booleans if provided.

Mapping from Spreadsheet Columns to JSON: - *Category* → `category` (string) in JSON. - *Sub Category* → `subcats[].name` in JSON. - *Item* → `items[].title` in JSON. - *Done* → `items[].done` (boolean). "TRUE"/"FALSE" in CSV becomes true/false. - *Subtask* → If not empty, it represents a subtask of the item. In JSON nested form, it becomes an object in `items[].subtasks` with `text` equal to the Subtask value. If Subtask is empty, the row describes the main item. - *Subtask Done* → `subtasks[].done` for that subtask (boolean). - *Subtask Hidden* (**new** CSV column) → `subtasks[].hidden` (boolean). - *ASVS* → `items[].asvs` (string). - *Description* → `items[].desc` (string). - *Tools* → `items[].tools` (string). - *Links* → `items[].links` (string). - *Applicability* → `items[].applic` (string). - *Sources* → `items[].sources` (string). - *Tags* → `items[].tags` (array of strings). In CSV/flat JSON this is a delimited string; our importer splits on commas/whitespace. For example `"auth session"` becomes `["auth","session"]`. (Empty or missing means no tags.) - *Priority* (**new** column) → `items[].priority` (integer). If omitted, default to 2 (medium). Non-numeric values (e.g. "High") are not expected in CSV; if present, the import could map "High"→3, etc., but we'll document to use 0–3. - *Hidden* (**new** column) → `items[].hidden` (boolean). "TRUE" means the item is hidden by default. - *Subcategory Notes* (**new** column) → `subcats[].notes` (string). As described, we expect this only on one row per subcategory (preferably the first item's row for that subcat). The importer will set the subcategory's notes when it encounters a non-empty value here. If multiple items in the subcategory have the same note text repeated, it will be recognized as duplicates (and not appended multiple times). If different text is encountered (which would likely be a user error), the importer might concatenate them with a newline, but ideally that situation is avoided. In practice, the exporter will output one consistent note on the first row.

Validation Rules & Edge Cases: - *Required fields:* In the nested JSON form, `category` (name) and subcategory `name` and item `title` are required. Others are optional and can be empty. In flat form, Category, Sub Category, and Item columns must be present (the importer will ignore rows missing these). - *Value ranges:* `priority` must be 0,1,2, or 3. If out of range or not an integer, we will clamp or default it (e.g., a priority "5" in imported data would be treated as 3, and negative or non-numeric would default to 2). The schema restricts 0–3. - *Boolean handling:* We accept various representations ("YES/NO", "1/0") as

synonyms for TRUE/FALSE in import [30]. Export will always use “TRUE”/“FALSE” for consistency. If an imported JSON uses actual booleans true/false (which is likely in a JSON file), our code will handle those (by conversion to string then lowercase, which works: e.g. true → "true"). - *String lengths/formats:* There are no strict length limits on text fields like Description or Notes – they can be multi-paragraph. However, for usability, extremely long text might be better kept in external docs with links. We encourage keeping notes concise. The Links field can contain URLs; we aren’t enforcing a URI format in validation because it might also include descriptive text. If needed, users should ensure links start with `http://` or `https://` (since we won’t execute them automatically, this is not strictly validated). - *Tags:* Should be simple keywords (no commas or whitespace within a single tag, since those act as separators). If a tag does contain a space or comma in a source (rare), the importer will split it – e.g. a tag value `"access control"` in the CSV would be split into `["access","control"]`. So users should use hyphen/underscore if they intend multi-word tags (“access-control”). This is a noted limitation of the simplistic splitting approach [20]. - *Duplication and Merge:* If the same item appears in multiple imported files (matching Category/Subcat/Item), the importer merges their fields (concatenating descriptions, merging tags, etc.) [19]. We should note that priority and hidden flags, if provided in multiple sources, the last read file would win (we don’t currently merge or average priorities – priority will simply be set when encountered; similarly, an item could be marked hidden in one file and not in another: the latest import’s value will apply). Typically, users will import from one master file so this is not an issue. - *Subcategory notes in merge:* As mentioned, if multiple rows supply a subcategory note, we handle the first and ignore or merge subsequent ones. We ensure not to duplicate the same note text. If two different notes were somehow provided for one subcategory (unexpected), our importer would currently append them separated by a newline (so the data isn’t lost). We assume users will avoid this scenario. - *Normalization:* We perform some trim and normalization (e.g., converting “Yes” to true, trimming spaces around fields). We also default missing fields to blank or false. This helps maintain a clean data state. - *Excel source mapping:* The original XLSX likely had columns mapping to these JSON fields (Category, Subcategory, Item, etc. as above). It may also have contained combined references (the `how_to_test` object in the transform code suggests the XLSX had structured content for tools and steps). In our JSON, we flatten those into the relevant text fields (e.g., if the Excel had separate columns for manual vs tool-based steps, those can be merged into the `tools` or `desc` field with headings). Our design assumes the data is now mostly consolidated into the fields we defined. Any structured data from the spreadsheet has been transformed into plain text for this app (for example, a list of tools becomes a newline-separated list in the Tools textarea).

In conclusion, the JSON schema above provides a clear and normalized structure for the checklist. It’s flexible enough to accommodate the new features (notes, priority, hidden) and can represent the entire spreadsheet content. The examples given show how real data populates this schema in both nested and flat forms.

Part C – Single-File HTML Prototype

Below is a **complete single-page HTML** file implementing the redesigned checklist application. **Instructions:** Save this as an `.html` file and open it in a desktop Chrome browser. It requires no server – all logic is client-side. You can test functionality by using the **Import Prefill** button to load a JSON/CSV (according to the formats described) or by editing the sample data already embedded. The prototype demonstrates collapsible categories, subcategory notes, priority coloring & filtering, hide/show hidden toggle, and inline editing of details, priority, and hidden flags. (For brevity, only a small sample dataset is embedded in `rawData` in this example. In a real scenario, you would import the full checklist CSV/JSON.)

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>Web App Security Checklist – Redesigned</title>
<style>
:root {
  --bg: #0b0f14;
  --fg: #e8f0ff;
  --muted: #9fb0c3;
  --card: #121822;
  --border: #1f2a37;
  --accent: #5fb3ff;
  --ok: #22c55e;   /* green (used for low priority perhaps) */
  --warn: #f59e0b; /* amber */
  --err: #ef4444;  /* red */
}
* { box-sizing: border-box; }
body {
  background: var(--bg); color: var(--fg);
  font-family: Inter, system-ui, Segoe UI, Roboto, Arial, sans-serif;
  margin: 0;
}
header {
  padding: 16px 18px;
  border-bottom: 1px solid var(--border);
  display: flex; gap: 12px; align-items: center; flex-wrap: wrap;
}
h1 { font-size: 1.25rem; margin: 0; }
.pill {
  background: #0f1722; border: 1px solid var(--border);
  padding: 6px 10px; border-radius: 999px;
  color: var(--muted); font-size: 0.9rem;
}
main { max-width: 1200px; margin: 0 auto; padding: 18px; }
.toolbar {
  display: flex; gap: 10px; flex-wrap: wrap; align-items: center;
  margin-bottom: 14px;
}
.toolbar input, .toolbar select, .toolbar button, label.button {
  background: #0e1622; color: var(--fg);
  border: 1px solid var(--border); border-radius: 10px;
  padding: 8px 12px; cursor: pointer;
  font: inherit;
}
.toolbar .small { font-size: 0.85rem; color: var(--muted); }
.grid {
```

```
  display: grid; gap: 14px;
  grid-template-columns: repeat(auto-fill, minmax(420px, 1fr));
}
.card {
  background: var(--card); border: 1px solid var(--border);
  border-radius: 14px; padding: 12px;
}
.card .top {
  display: flex; align-items: center; justify-content: space-between;
  gap: 10px; cursor: pointer;
}
.card.collapsed .card-content { display: none; }
.card .title { font-weight: 700; font-size: 1.05rem; }
.card .count { color: var(--muted); font-size: 0.85rem; }
.subcat {
  margin-top: 10px; padding-top: 6px;
  border-top: 1px dashed var(--border);
}
.subcat h3 {
  margin: 6px 0 8px; font-size: 0.95rem; color: #cfe4ff;
  display: flex; align-items: center; gap: 6px;
}
.subcat h3 input[type="checkbox"] { transform: scale(1.1); margin-right: 4px; }
.subcat-details summary {
  cursor: pointer; color: var(--accent);
  font-weight: 600; margin-top: 4px;
}
.item {
  background: #0c1420; border: 1px solid var(--border);
  border-radius: 10px; padding: 8px;
  margin-bottom: 8px;
  position: relative;
}
.item-head { display: flex; align-items: center; gap: 8px; }
.item-head input[type="checkbox"] { transform: scale(1.15); }
.item-title { flex: 1; }
.item-meta {
  color: var(--muted); font-size: 0.85rem;
  margin-top: 4px;
}
.tags {
  display: flex; gap: 6px; flex-wrap: wrap;
  margin: 4px 0 6px 0;
}
.tag {
  font-size: 0.75rem; padding: 2px 8px;
  border: 1px solid var(--border); border-radius: 999px;
  background: #0b1a2a; color: #b8cff0;
```

```
  cursor: pointer;
}
.subtasks {
  margin-top: 6px; padding-left: 14px;
  border-left: 2px solid #1e293b;
}
.subtask {
  display: flex; align-items: center; gap: 6px;
  margin: 4px 0;
}
.subtask input[type="checkbox"] { transform: scale(1.05); }
.hidden-item { opacity: 0.6; }       /* Hidden items appear faded when shown */
.hidden-subtask { opacity: 0.6; font-style: italic; }  /* Hidden subtask style
*/
.item.priority-3 { border-left: 4px solid var(--err); }
.item.priority-2 { border-left: 4px solid var(--warn); }
.item.priority-1 { border-left: 4px solid var(--ok); opacity: 0.85; } /* low:
green-ish, slightly faded */
.item.priority-0 { border-left: 4px solid #6b7280; } /* very low: gray */
details summary {
  cursor: pointer; color: var(--accent);
  font-weight: 600; margin-top: 8px;
}
textarea, select {
  width: 100%;
  background: #0b111a; color: var(--fg);
  border: 1px solid var(--border); border-radius: 8px;
  padding: 6px 8px; margin-top: 4px;
  font: inherit;
}
.item-meta select {
  background: #0b111a; color: var(--fg);
  border: 1px solid var(--border); border-radius: 6px;
  padding: 4px 6px; margin-left: 6px;
}
.item-meta input[type="checkbox"] {
  transform: scale(1.0); margin-right: 4px;
  accent-color: var(--accent);
}
footer {
  color: var(--muted); text-align: center; padding: 12px;
  font-size: 0.8rem;
}
</style>
</head>
<body>
<header>
  <h1>Web App Security Checklist – Category/Subtask View</h1>
```

```
  <span class="pill" id="overall">0/0 checks completed</span>
  <span class="pill" id="catcount">0 categories</span>
  <span class="pill">Local-only • saves in your browser</span>
</header>
<main>
  <div class="toolbar">
    <input id="search" type="text" placeholder="Search..." />
    <select id="filter">
      <option value="all">All</option>
      <option value="open">Open</option>
      <option value="done">Completed</option>
    </select>
    <!-- Priority filter checkboxes -->
    <span class="small">Priority:</span>
    <label class="small"><input type="checkbox" class="prioChk" value="3"
checked /> High</label>
    <label class="small"><input type="checkbox" class="prioChk" value="2"
checked /> Med</label>
    <label class="small"><input type="checkbox" class="prioChk" value="1"
checked /> Low</label>
    <label class="small"><input type="checkbox" class="prioChk" value="0"
checked /> Very Low</label>
    <label class="small"><input id="showHidden" type="checkbox" /> Show hidden</
label>
    <!-- Action buttons -->
    <button id="save">Save (local)</button>
    <button id="exportJSON">Export JSON</button>
    <button id="exportCSV">Export CSV</button>
    <button id="reset">Reset (from file)</button>
    <label for="fileInput" class="button">Import Prefill (CSV/JSON)</label>
    <input id="fileInput" type="file" accept=".csv,.json"
style="display:none;" />
  </div>
  <div class="small" style="margin-bottom: 10px;">
    <strong>Tip:</strong> You can import a <b>CSV</b> or <b>JSON</b> file to
prefill data.
    CSV header should include:
    <code>Category, Sub Category, Item, Done, Subtask, Subtask Done, Subtask
Hidden, ASVS, Description, Tools, Links, Applicability, Sources, Tags, Priority,
Hidden, Subcategory Notes</code>.
  </div>
  <div id="grid" class="grid"></div>
  <p style="font-size:0.85rem; color: var(--muted);">
    Each category card contains sub-categories, which contain items (with
checkboxes). Items may have sub-tasks (nested checkboxes) and an expandable
<em>Details</em> section (ASVS, description, etc.). Use the filters above to
narrow the list by text, completion status, or priority. Hidden items are
omitted by default (enable "Show hidden" to view them). Your changes auto-save
```

```
locally; use Export to save your progress externally.
  </p>
</main>
<footer>Offline checklist tool – no server required. Data stays in your browser
(export to share).</footer>

<script>
/*** Data Initialization ***/
const STORAGE_KEY = "webchecklist_v1.0";
const rawData = [
  { category: "Category A",
    subcats: [
      { name: "Subcat A1", notes: "",
        items: [
          { title: "Visible Item", done: false, hidden: false, priority: 3,
            asvs: "", desc: "Example item (high priority).", tools: "", links:
"", applic: "", sources: "",
            tags: ["auth","session"], subtasks: []
          }
        ]
      },
      { name: "Subcat A2", notes: "This subcategory has a note.",
        items: [
          { title: "Hidden Item", done: false, hidden: true, priority: 1,
            asvs: "", desc: "This is a low-priority item hidden by default.",
tools: "", links: "", applic: "", sources: "",
            tags: [], subtasks: []
          }
        ]
      }
    ]
  },
  { category: "Category B",
    subcats: [
      { name: "Subcat B1", notes: "",
        items: [
          { title: "Item with Subtasks", done: false, hidden: false, priority:
2,
            asvs: "", desc: "An item with one hidden subtask.", tools: "",
links: "", applic: "", sources: "",
            tags: ["files"], subtasks: [
              { text: "Visible subtask", done: false, hidden: false },
              { text: "Hidden subtask", done: false, hidden: true }
            ]
          }
        ]
      }
    ]
```

```
  }
];
function loadData() {
  try {
    const saved = localStorage.getItem(STORAGE_KEY);
    const data = saved ? JSON.parse(saved) : rawData;
    return Array.isArray(data) ? data : rawData;
  } catch (e) {
    console.warn("Failed to parse saved data, resetting.", e);
    return rawData;
  }
}
function saveData() {
  localStorage.setItem(STORAGE_KEY, JSON.stringify(data));
}

/*** State Variables ***/
let data = loadData();
// Ensure new fields exist with defaults for all items/subtasks:
data.forEach(cat => {
  cat.subcats?.forEach(sc => {
    sc.notes = sc.notes || "";
    sc.items?.forEach(it => {
      if (it.hidden === undefined) it.hidden = false;
      if (it.priority === undefined) it.priority = 2;
      it.tags = Array.isArray(it.tags) ? it.tags : (it.tags ?
String(it.tags).split(/[\\s,;]+/).filter(Boolean) : []);
      it.subtasks?.forEach(st => {
        if (st.hidden === undefined) st.hidden = false;
      });
    });
  });
});
let tagFilter = "";               // current tag filter (one tag at a time)
let showHidden = false;          // whether to show hidden items
const prioFilterSet = new Set([0,1,2,3]);  // which priority levels are shown

/*** Helper Functions ***/
const $ = sel => document.querySelector(sel);
function createElem(tag, props = {}, ...children) {
  const el = document.createElement(tag);
  Object.assign(el, props);
  for (let child of children) {
    if (typeof child === "string") {
      el.append(document.createTextNode(child));
    } else if (child) {
      el.append(child);
    }
```

```
  }
  return el;
}
// Utility: parse CSV text into array of row objects
function csvToObjects(csvText) {
  const lines = csvText.split(/\\r?\\n/).filter(l => l.trim().length);
  if (!lines.length) return [];
  const headers = lines[0].split(/,(?=(?:[^"]*\\"[^"]*\\")*[^"]*$)/)
                          .map(h => h.replace(/^"|"$/g, "").trim());
  const rows = [];
  for (let i = 1; i < lines.length; i++) {
    const cols = lines[i].split(/,(?=(?:[^"]*\\"[^"]*\\")*[^"]*$)/)
                         .map(c => c.replace(/^"|"$/g, ""));
    const obj = {};
    headers.forEach((h, idx) => { obj[h] = cols[idx] !== undefined ?
cols[idx] : ""; });
    rows.push(obj);
  }
  return rows;
}

/*** Import Merge Logic ***/
function ensureCategory(dataArr, catName) {
  const name = (catName || "").trim();
  if (!name) return null;
  let cat = dataArr.find(c => c.category.toLowerCase() === name.toLowerCase());
  if (!cat) {
    cat = { category: name, subcats: [] };
    dataArr.push(cat);
  }
  return cat;
}
function ensureSubcat(catObj, subName) {
  const name = (subName || "(General)").trim() || "(General)";
  let sc = catObj.subcats.find(s => s.name.toLowerCase() ===
name.toLowerCase());
  if (!sc) {
    sc = { name: name, notes: "", items: [] };
    catObj.subcats.push(sc);
  }
  return sc;
}
function ensureItem(subcatObj, itemTitle) {
  const title = (itemTitle || "(no title)").trim() || "(no title)";
  let it = subcatObj.items.find(it => it.title.toLowerCase() ===
title.toLowerCase());
  if (!it) {
    it = { title: title, done: false, hidden: false, priority: 2,
```

```
            asvs: "", desc: "", tools: "", links: "", applic: "", sources: "",
            tags: [], subtasks: [] };
    subcatObj.items.push(it);
  }
  return it;
}
function mergePrefillRows(rows) {
  rows.forEach(row => {
    const cat = ensureCategory(data, row["Category"]);
    if (!cat) return;
    const sc = ensureSubcat(cat, row["Sub Category"]);
    const it = ensureItem(sc, row["Item"]);
    // Main item done status
    const doneVal = (row["Done"] || "").toString().trim().toLowerCase();
    if (doneVal === "true" || doneVal === "1" || doneVal === "yes") {
      it.done = true;
    } else if (doneVal === "false" || doneVal === "0" || doneVal === "no") {
      it.done = false;
    }
    // Subtask (if present)
    const subtaskText = (row["Subtask"] || "").trim();
    if (subtaskText) {
      const subDoneVal = (row["Subtask Done"] ||
"").toString().trim().toLowerCase();
      const subHiddenVal = (row["Subtask Hidden"] ||
"").toString().trim().toLowerCase();
      const stDone = (subDoneVal === "true" || subDoneVal === "1" || subDoneVal
=== "yes");
      const stHidden = (subHiddenVal === "true" || subHiddenVal === "1" ||
subHiddenVal === "yes");
      // find or add subtask
      it.subtasks = it.subtasks || [];
      const existingSt = it.subtasks.find(s => s.text === subtaskText);
      if (!existingSt) {
        it.subtasks.push({ text: subtaskText, done: stDone, hidden: stHidden });
      } else {
        // If it exists, optionally update its done/hidden if provided
        if (existingSt.done !== stDone) existingSt.done = stDone;
        if (existingSt.hidden !== stHidden) existingSt.hidden = stHidden;
      }
    }
    // Merge detail fields (append if existing content)
    if (row["ASVS"]) {
      it.asvs = it.asvs ? it.asvs + "\\n" + row["ASVS"] : row["ASVS"];
    }
    if (row["Description"]) {
      it.desc = it.desc ? it.desc + "\\n\\n" + row["Description"] :
row["Description"];
```

```
    }
    if (row["Tools"]) {
      it.tools = it.tools ? it.tools + "\\n\\n" + row["Tools"] : row["Tools"];
    }
    if (row["Links"]) {
      it.links = it.links ? it.links + "\\n\\n" + row["Links"] : row["Links"];
    }
    if (row["Applicability"]) {
      it.applic = it.applic ? it.applic + "\\n\\n" + row["Applicability"] :
row["Applicability"];
    }
    if (row["Sources"]) {
      it.sources = it.sources ? it.sources + "\\n\\n" + row["Sources"] :
row["Sources"];
    }
    // Tags (split and merge)
    if (row["Tags"]) {
      const tags = row["Tags"].split(/[;,\\s]+/).map(t =>
t.trim()).filter(Boolean);
      it.tags = Array.from(new Set([...(it.tags||[]), ...tags]));
    }
    // Priority
    if (row["Priority"] !== undefined && row["Priority"] !== "") {
      const pr = row["Priority"];
      let prNum = (typeof pr === "number") ? pr : parseInt(pr);
      if (!isNaN(prNum)) {
        prNum = Math.max(0, Math.min(3, prNum));
        it.priority = prNum;
      } else {
        // accept textual priorities like "High"/"Low" if present
        const pText = pr.toString().toLowerCase();
        if (pText.includes("high")) it.priority = 3;
        else if (pText.includes("medium")) it.priority = 2;
        else if (pText.includes("low")) it.priority = 1;
        else if (pText.includes("very low") || pText.includes("very-low"))
it.priority = 0;
      }
    }
    // Hidden
    if (row["Hidden"] !== undefined && row["Hidden"] !== "") {
      const hidVal = row["Hidden"].toString().trim().toLowerCase();
      if (hidVal === "true" || hidVal === "1" || hidVal === "yes") {
        it.hidden = true;
      } else if (hidVal === "false" || hidVal === "0" || hidVal === "no") {
        it.hidden = false;
      }
    }
    // Subcategory Notes
```

```
    if (row["Subcategory Notes"]) {
      const note = row["Subcategory Notes"].trim();
      if (note) {
        if (!sc.notes || !sc.notes.trim()) {
          sc.notes = note;
        } else if (sc.notes.trim() !== note) {
          // If a different note exists, append (edge case)
          sc.notes += "\\n" + note;
        }
      }
    }
  });
  saveData();
  render();
}

/*** Export Functions ***/
function exportToCSV(dataArr) {
  const header = ["Category","Sub Category","Item","Done",
                  "Subtask","Subtask Done","Subtask Hidden",

"ASVS","Description","Tools","Links","Applicability","Sources","Tags",
                  "Priority","Hidden","Subcategory Notes"];
  const rows = [header];
  dataArr.forEach(cat => {
    cat.subcats.forEach(sc => {
      let subcatNoteOutput = false;
      sc.items.forEach((it, itemIndex) => {
        // Helper to output a row (either item or subtask row)
        const outputRow = (subText, subDone, subHidden, noteVal) => {
          rows.push([
            cat.category,
            sc.name,
            it.title,
            it.done ? "TRUE" : "FALSE",
            subText,
            subDone,
            subHidden,
            it.asvs || "",
            it.desc || "",
            it.tools || "",
            it.links || "",
            it.applic || "",
            it.sources || "",
            (it.tags||[]).join(" "),
            it.priority !== undefined ? String(it.priority) : "",
            it.hidden ? "TRUE" : "FALSE",
            noteVal
```

```
          ]);
        };
        if (Array.isArray(it.subtasks) && it.subtasks.length > 0) {
          it.subtasks.forEach((st, stIndex) => {
            // Only include subcategory note on the *first* output row of this
subcategory
            let noteVal = "";
            if (!subcatNoteOutput) {
              noteVal = sc.notes || "";
              subcatNoteOutput = true;
            }
            outputRow(st.text, st.done ? "TRUE":"FALSE", st.hidden ?
"TRUE":"FALSE", noteVal);
          });
        } else {
          // No subtask row; output the item itself
          let noteVal = "";
          if (!subcatNoteOutput) {
            noteVal = sc.notes || "";
            subcatNoteOutput = true;
          }
          outputRow("", "", "", noteVal);
        }
      });
    });
  });
  // Escape and quote each field, then join
  return rows.map(cols => cols.map(val => {
    const str = val == null ? "" : String(val);
    // escape double quotes by doubling them
    return `"${str.replace(/"/g, '""')}"`;
  }).join(",")).join("\\n");
}

/*** Rendering ***/
function render() {
  const filterText = $("#search").value.toLowerCase().trim();
  const statusFilter = $("#filter").value;  // "all", "open", or "done"
  const grid = $("#grid");
  grid.innerHTML = "";  // clear current view
  let totalCount = 0, doneCount = 0, categoryCount = 0;
  data.forEach(cat => {
    let catTotal = 0, catDone = 0;
    const card = createElem("div", { className: "card" });
    // Category header with master checkbox
    const catCheckbox = createElem("input", { type: "checkbox", checked:
false });
    // When category checkbox toggled, mark all items in category done/undone
```

```
    catCheckbox.addEventListener("change", () => {
      cat.subcats.forEach(sc => {
        sc.items.forEach(it => { it.done = catCheckbox.checked; });
      });
      updateCounts();
    });
    const titleEl = createElem("div", { className: "title" }, cat.category ||
"(no category)");
    const countEl = createElem("div", { className: "count" }, "0/0");
    const topBar = createElem("div", { className: "top" }, catCheckbox,
titleEl, countEl);
    // Click on category bar toggles collapse (ignore clicks on the checkbox
itself)
    topBar.addEventListener("click", e => {
      if (e.target !== catCheckbox) {
        card.classList.toggle("collapsed");
      }
    });
    const contentWrapper = createElem("div", { className: "card-content" });
    // Subcategories
    cat.subcats.forEach(sc => {
      const secDiv = createElem("div", { className: "subcat" });
      // Subcategory header with checkbox
      const subcatCheckbox = createElem("input", { type: "checkbox", checked:
false });
      subcatCheckbox.addEventListener("change", () => {
        sc.items.forEach(it => { it.done = subcatCheckbox.checked; });
        updateCounts();
      });
      const subcatHeader = createElem("h3", {}, subcatCheckbox, sc.name ||
"(General)");
      secDiv.append(subcatHeader);
      // We'll append notes and items after filtering
      let anyItemShown = false;
      sc.items.forEach(it => {
        // Filter conditions:
        if (filterText) {
          // Build a haystack of searchable text: category + subcat + item title
+ tags + description + tools + links
          const hay = (cat.category + " " + sc.name + " " + it.title + " " +
                       (it.tags || []).join(" ") + " " + (it.asvs || "") + " " +
                       (it.desc || "") + " " + (it.tools || "") + " " +
(it.links || "")).toLowerCase();
          if (!hay.includes(filterText)) return;  // text filter not matched
        }
        if (tagFilter && !(it.tags || []).includes(tagFilter)) return;  // tag
filter active and item doesn't have it
        if (!showHidden && it.hidden) return;  // hidden items filtered out
```

```
        if (statusFilter === "open" && it.done) return;    // skip done items
        if (statusFilter === "done" && !it.done) return;   // skip open items
        if (!prioFilterSet.has(it.priority ?? 2)) return;  // skip if priority
not selected
        // If we reach here, item should be shown
        anyItemShown = true;
        const itemDiv = createElem("div", { className: "item" });
        // Add priority and hidden CSS classes
        itemDiv.classList.add(`priority-${it.priority !== undefined ?
it.priority : 2}`);
        if (it.hidden) itemDiv.classList.add("hidden-item");
        // Item header with checkbox and title
        const itemHead = createElem("div", { className: "item-head" });
        const itemCb = createElem("input", { type: "checkbox", checked: !!
it.done });
        itemCb.addEventListener("change", () => {
          it.done = itemCb.checked;
          updateCounts();
        });
        itemHead.append(itemCb, createElem("div", { className: "item-title" },
it.title || "(no title)"));
        itemDiv.append(itemHead);
        // Tags
        if (it.tags && it.tags.length) {
          const tagsDiv = createElem("div", { className: "tags" });
          it.tags.forEach(tag => {
            const tagSpan = createElem("span", { className: "tag" }, tag);
            tagSpan.addEventListener("click", () => {
              tagFilter = (tagFilter === tag ? "" : tag);
              render();
            });
            tagsDiv.append(tagSpan);
          });
          itemDiv.append(tagsDiv);
        }
        // Subtasks (if any)
        if (Array.isArray(it.subtasks) && it.subtasks.length > 0) {
          const stContainer = createElem("div", { className: "subtasks" });
          it.subtasks.forEach(st => {
            if (!showHidden && st.hidden) return; // hide hidden subtask unless
toggled
            // (We don't filter subtasks by priority or search text separately –
assume they fall under item's filtering)
            const stRow = createElem("div", { className: "subtask" });
            if (st.hidden) stRow.classList.add("hidden-subtask");
            const stCb = createElem("input", { type: "checkbox", checked: !!
st.done });
            stCb.addEventListener("change", () => {
```

```
            st.done = stCb.checked;
            updateCounts();
          });
          stRow.append(stCb, createElem("div", {}, st.text));
          stContainer.append(stRow);
        });
        if (stContainer.children.length > 0) {
          itemDiv.append(stContainer);
        }
      }
      // Details section (collapsible)
      const detailsEl = createElem("details", {});
      const summaryEl = createElem("summary", {}, "Details");
      detailsEl.append(summaryEl);
      // Priority selector
      const prioLabel = createElem("label", { className: "item-meta" });
      const prioSelect = createElem("select", {});
      ["3 - High","2 - Medium","1 - Low","0 - Very Low"].forEach(optionText
=> {
        const opt = createElem("option", {});
        opt.value = optionText[0];  // first char is the number
        opt.textContent = optionText;
        if (parseInt(opt.value) === (it.priority ?? 2)) {
          opt.selected = true;
        }
        prioSelect.append(opt);
      });
      prioSelect.addEventListener("change", () => {
        it.priority = parseInt(prioSelect.value);
        saveData();
        render();  // re-render to update item order/color if needed
      });
      prioLabel.append("Priority: ", prioSelect);
      detailsEl.append(prioLabel);
      // Hide toggle
      const hideLabel = createElem("label", { className: "item-meta" });
      const hideCheck = createElem("input", { type: "checkbox", checked: !!
it.hidden });
      hideCheck.addEventListener("change", () => {
        it.hidden = hideCheck.checked;
        saveData();
        render();
      });
      hideLabel.append(hideCheck, " Hide by default");
      detailsEl.append(hideLabel);
      // Editable text fields (ASVS, Description, etc.)
      const metaFields = [
        ["ASVS", it.asvs || ""],
```

```
          ["Description", it.desc || ""],
          ["Tools", it.tools || ""],
          ["Links", it.links || ""],
          ["Applicability / N/A", it.applic || ""],
          ["Sources", it.sources || ""]
        ];
        metaFields.forEach(([label, text]) => {
          detailsEl.append(createElem("div", { className: "item-meta" },
label));
          const textarea = createElem("textarea", {});
          textarea.value = text;
          textarea.addEventListener("input", () => {
            // Save changes to corresponding field
            switch(label) {
              case "ASVS": it.asvs = textarea.value; break;
              case "Description": it.desc = textarea.value; break;
              case "Tools": it.tools = textarea.value; break;
              case "Links": it.links = textarea.value; break;
              case "Applicability / N/A": it.applic = textarea.value; break;
              case "Sources": it.sources = textarea.value; break;
            }
            saveData();
          });
          detailsEl.append(textarea);
        });
        itemDiv.append(detailsEl);
        secDiv.append(itemDiv);
        // Update counters
        catTotal++;
        if (it.done) catDone++;
        totalCount++;
        if (it.done) doneCount++;
      }); // end items loop
      // Append subcategory notes section if anyItemShown and if notes exist or
we want to allow adding notes
      if (anyItemShown) {
        // Insert subcategory notes details after header
        const notesDetails = createElem("details", { className: "subcat-
details" });
        const notesSummary = createElem("summary", {}, "Notes");
        notesDetails.append(notesSummary);
        const notesArea = createElem("textarea", {});
        notesArea.value = sc.notes || "";
        notesArea.addEventListener("input", () => {
          sc.notes = notesArea.value;
          saveData();
        });
        notesDetails.append(notesArea);
```

```
        // If there's no note content, we could leave it closed by default
(details default is closed anyway).
        secDiv.insertBefore(notesDetails, secDiv.children[1] || null);
        contentWrapper.append(secDiv);
      }
    }); // end subcats loop
    if (catTotal > 0) {
      // Update category completion count in header
      countEl.textContent = `${catDone}/${catTotal}`;
      card.append(topBar, contentWrapper);
      grid.append(card);
      categoryCount++;
      // Mark category checkbox checked state if all items done (and not zero
items)
      catCheckbox.checked = (catDone === catTotal);
    }
  }); // end categories loop
  // Update overall counters
  $("#overall").textContent = `${doneCount}/${totalCount} checks completed`;
  $("#catcount").textContent = `${categoryCount} categories`;
}
function updateCounts() {
  saveData();
  render();
}

/*** Event Listeners for Filters and Buttons ***/
$("#search").addEventListener("input", () => { render(); });
$("#filter").addEventListener("change", () => { render(); });
document.querySelectorAll(".prioChk").forEach(cb => {
  cb.addEventListener("change", e => {
    const val = parseInt(e.target.value);
    if (e.target.checked) {
      prioFilterSet.add(val);
    } else {
      prioFilterSet.delete(val);
    }
    render();
  });
});
$("#showHidden").addEventListener("change", e => {
  showHidden = e.target.checked;
  render();
});
$("#save").addEventListener("click", () => { saveData(); alert("Data saved
locally."); });
$("#reset").addEventListener("click", () => {
  // Reset to embedded rawData snapshot
```

```
  data = JSON.parse(JSON.stringify(rawData));
  saveData();
  render();
});
$("#exportJSON").addEventListener("click", () => {
  const blob = new Blob([JSON.stringify(data, null, 2)], { type: "application/
json" });
  const a = document.createElement("a");
  a.href = URL.createObjectURL(blob);
  a.download = "webapp_checklist.json";
  document.body.appendChild(a);
  a.click();
  a.remove();
});
$("#exportCSV").addEventListener("click", () => {
  const csvStr = exportToCSV(data);
  const blob = new Blob([csvStr], { type: "text/csv" });
  const a = document.createElement("a");
  a.href = URL.createObjectURL(blob);
  a.download = "webapp_checklist.csv";
  document.body.appendChild(a);
  a.click();
  a.remove();
});
$("#fileInput").addEventListener("change", event => {
  const file = event.target.files[0];
  if (!file) return;
  const reader = new FileReader();
  reader.onload = e => {
    try {
      const content = e.target.result;
      const name = file.name.toLowerCase();
      if (name.endsWith(".json")) {
        const obj = JSON.parse(content);
        if (Array.isArray(obj) && obj.length && typeof obj[0] === "object") {
          if ("category" in obj[0] && "items" in obj[0] && !("subcats" in
obj[0])) {
            // Form 1: array of categories with flat items list
            // (This form might not be used often; but we can transform using
logic similar to transformImportedData)
            data = obj.map(oldCat => {
              const newCat = { category: oldCat.category, subcats: [] };
              // The old structure possibly had category with an 'items' array
flat and some grouping by ASVS parts
              // For simplicity, we can push all items under a default
subcategory.
              const defaultSub = { name: "(General)", notes: "", items: [] };
              (oldCat.items || []).forEach(item => {
```

```
                const newItem = {
                  title: item.title || "(no title)",
                  done: false,
                  hidden: false,
                  priority: 2,
                  asvs: item.asvs || "",
                  desc: item.simple_desc || item.desc || "",
                  tools: item.tools || "",
                  links: item.references || item.links || "",
                  applic: item.applicability || "",
                  sources: (Array.isArray(item.sources) ? item.sources.join("\
\n") : item.sources) || "",
                  tags: item.tags || [],
                  subtasks: []
                };
                // If the source format had subtasks or additional fields,
handle as needed (not detailed here).
                defaultSub.items.push(newItem);
              });
              newCat.subcats.push(defaultSub);
              return newCat;
            });
          } else if ("category" in obj[0] && ("subcats" in obj[0] ||
"subCategories" in obj[0])) {
            // Form 2: Already nested full dataset form
            data = obj;
          } else if (Array.isArray(obj)) {
            // Form 3: flat array of row objects
            data = loadData(); // start from current data
            mergePrefillRows(obj);
            return; // mergePrefillRows calls render
          }
          saveData();
          render();
        } else {
          alert("Unrecognized JSON format. Please provide an array of
objects.");
        }
      } else if (name.endsWith(".csv")) {
        const rows = csvToObjects(content);
        mergePrefillRows(rows);
      } else {
        alert("Unsupported file type. Please import a .csv or .json file.");
      }
    } catch (err) {
      console.error("Import failed:", err);
      alert("Failed to import: " + err.message);
    }
```

```
  };
  reader.readAsText(file);
});

/*** Initial Render ***/
render();
</script>
</body>
</html>
```

Usage Notes: When you open this file, it will load with some example data (two categories, demonstrating priority colors and hidden items). You can try toggling the "Show hidden" checkbox to see hidden items appear/disappear (in the sample, "Hidden Item" in Category A2 is hidden by default). Try changing the priority filter checkboxes – unchecking "Low" will remove that hidden item (priority 1) even if "Show hidden" is on, etc. The counters at the top update as you check items or subtasks. The Import Prefill button accepts a CSV or JSON; after import, your data is merged or replaced accordingly. The **Export CSV/JSON** buttons will download the current data with the new schema (including the new fields where applicable). The file is designed for **Chrome desktop**; other modern browsers should work too, but we have not optimized for mobile or older IE, etc.

Part D – Lightweight Library Assessment

Our implementation uses vanilla JavaScript for everything. We considered a few optional libraries that could enhance functionality or developer experience, along with their trade-offs:

Library	Purpose	Size (minified)	License	Pros / Cons
Papa Parse	CSV parsing (in-browser)	~20 KB	MIT	**Pros:** Very fast and robust CSV parser, handles edge cases (quoted fields, big files) and can stream large files. **Cons:** Our custom parser (~20 lines) already handles the simple CSV schema we have (including quoted commas), and our data sizes are not huge. Including PapaParse would add overhead for little gain in our scenario.

Library	Purpose	Size (minified)	License	Pros / Cons
SheetJS (xlsx)	Read Excel (.xls/.xlsx)	~70–100 KB	Apache-2.0	**Pros:** Allows direct import of the Excel spreadsheet without conversion to CSV. Full support for Excel formats. **Cons:** Quite large for a single-file tool, and not needed if users can save as CSV. Would increase complexity (binary parsing) and memory use. Could be offered as an optional separate import path if demand arises, but not included by default to keep this file lean.
Clusterize.js	List virtualization for large DOMs	~4 KB	MIT	**Pros:** Efficiently handles very long lists by only rendering visible items, improving performance if thousands of DOM nodes would be present. Very small footprint and vanilla JS. **Cons:** Adds complexity – our list is hierarchical (categories, subcats, items), which is harder to virtualize compared to a flat list. Not necessary unless the checklist grows extremely large. For current use (~hundreds of items), performance is acceptable without it.
DOMPurify	HTML sanitization (security)	~8 KB	Apache-2.0	**Pros:** If we allowed rich HTML input (e.g., user could style descriptions or we make links clickable), DOMPurify would neutralize any malicious content (XSS). **Cons:** Not needed currently since we're not injecting any HTML from users – all inputs are treated as plain text. Including it now would be extra payload with no effect unless we change requirements.

We decided **not** to include any of these by default. The base application meets requirements without them: our CSV parser is sufficient, and users can convert Excel to CSV externally. Should the dataset expand dramatically or richer text features be needed, we could integrate these libraries in a future phase. Notably, all these libraries are MIT/Apache licensed, meaning they are compatible with an open-source project. If needed, they could be loaded via CDN `<script>` tags (since we want to avoid a build process). For example, Papa Parse could be added via CDN and used to stream parse a huge CSV if performance became an issue. But for now, simplicity wins.

Part E – Roadmap & Future Improvements

Phase 1 – MVP Implementation: (Completed in the prototype above) Focus on integrating the new data fields and UI features: - Add **Priority and Hidden fields** to data model; update import/export logic for CSV/JSON to handle them. - **UI controls:** Priority filter checkboxes, Show Hidden toggle, subcategory Notes field, and inline priority/hidden editors in item details. - Ensure all basic features work together: filtering by text/tag/status/priority, hide/show logic, and persistence. - Testing with sample data to confirm that hidden items stay hidden, priority colors render correctly, and CSV/JSON round-trip includes the new fields.

Phase 2 – Refinement and Usability: - **Polish UI:** Adjust styling based on user feedback (e.g., tweak priority colors if needed for better contrast on dark background, perhaps add legends or tooltips for what each color means). We might introduce a small legend in the footer or header explaining the color code (e.g., a colored square with "High Priority" label). - **Accessibility & UX:** Although mobile and a11y were out of scope, if time permits, ensure keyboard navigation is possible (currently checkboxes and details are focusable). Possibly increase text contrast for readability. Also consider remembering detail expansion state – right now, every re-render closes all expanded detail panels, which could be annoying. We could preserve open panels by, for example, tracking expanded item IDs and re-opening them after render, or by avoiding full re-render on minor edits (update DOM in place). - **Validation & Error Handling:** Add more robust file import validation. For example, if a CSV is missing required columns or has typos in headers, show a warning (currently we assume correct headers). If JSON import is not in an expected form, we attempt to detect it but could give more user-friendly messages. This aligns with the README "lightweight validation during import" goal [31]. - **Performance Tuning:** If users report slowness with large datasets, implement optimizations. For instance, use `documentFragment` to batch DOM updates (though current code already builds elements in memory then appends, which is fairly optimal). Possibly implement on-demand rendering for collapsed categories (don't render their items until expanded). Another idea is to only render items matching the current filter (which we do) – if a user searches, we already skip non-matching items which is good. For extremely large category counts, a search-as-you-type might still be heavy; we could throttle the search input handler. These changes would ensure the app remains snappy as the content scales up.

Phase 3 – Extended Features and Packaging: - **Full Data Population:** Work with the master spreadsheet data to populate all categories and items. This is more content work than coding – converting the XLSX into our JSON/CSV format. Once populated, test the app's performance and use patterns with the full list (~the OWASP WSTG + other sources as mentioned). - **Related Items by Tag:** A potential nice-to-have (mentioned in README) is showing related items in the Details pane based on common tags [32]. We could implement this by, say, when a Details is open, listing other item titles that share a tag. This would help users navigate between related tests (e.g., items tagged "injection" might be scattered in different categories). We'd need to decide how to display these (maybe a small list of links in the Details section). - **Category-level ASVS mapping:** Another future addition could be an ASVS field at category level [33]. If the source data provides an ASVS chapter mapping per category, we could add a text field at the top of each category card for "Category ASVS refs". - **Team Collaboration:** While a truly multi-user version would require a server (out of scope), we could facilitate manual collaboration by adding an "Import from JSON (merge)" vs "Replace" option, or providing a way to highlight which items changed since last export. Another idea: an **"Export Report"** function that generates a Markdown or PDF summary of completed items. The README hints at exporting to Markdown for report writing [34] – we could add a button to each item's Details to copy a Markdown snippet of that item (including its notes and status). - **Modular Build (Optional):** If the code grows much further, we might break it into logical modules (e.g., separate JS for import/export logic,

rendering logic, etc.) and use a build step to combine them into one file. We could also host it as a GitHub page or similar for easy access. However, we'd still provide a single HTML for offline use. Modernizing the code with ES6 modules could improve organization, but we chose to keep everything in one `<script>` to adhere to the single-file requirement. In a future refactor, we might separate out the style and script for maintainability, while providing a build script to inline them for distribution (ensuring the end-user still gets a one-file solution). - **Testing:** Develop a small test suite for critical functions like `csvToObjects`, `exportToCSV`, and the merge logic. This could be simple console-based tests or using a framework like Mocha in the browser. Given this is an offline tool, formal testing might be limited, but at least having sample imports/exports to verify no data loss (especially with new fields) is important as we iterate.

Maintenance & Support: The app, as redesigned, remains straightforward to maintain. Content updates (adding new checklist items) can be done via CSV and imported – no code changes needed. The focus should be on ensuring the data model stays aligned if new kinds of data are added (for example, if one wanted to add a "Category" field at the item level mapping to ASVS chapters, we'd extend similarly). We have future-proofed by making the import and export adapt to added fields gracefully (unknown fields would be preserved in row objects but perhaps not used – currently we ignore any extra columns we don't recognize in `mergePrefillRows`). In the roadmap above, we align future features with what was mentioned in the original Next Steps, ensuring our design can accommodate them.

Conclusion: This redesign improves the usability of the Web App Security Checklist by introducing priority-based focusing and optional items, without compromising the simplicity of a single-file, client-only app. We've addressed known gaps (filter toggles, data merging issues) and set the stage for easy content growth. Users can now customize their view to concentrate on high-priority tests and hide noise, all while having the ability to document notes at various levels. The security of the approach remains solid (no eval or DOM XSS vulnerabilities introduced), and performance will scale to the medium dataset anticipated. Further improvements can be layered on as needed, following the roadmap above, making this tool a robust companion for web application security testing.

---

1 2 3 4 5 6 7 28 29 31 32 33 34 README.md
https://github.com/cornish337/WebCheckSec/blob/f951aeaf0cfd58c0443eee2e7ea2b28de7d0f739/README.md

8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 30
webapp_security_checklist_example.html
https://github.com/cornish337/WebCheckSec/blob/f951aeaf0cfd58c0443eee2e7ea2b28de7d0f739/webapp_security_checklist_example.html