Home Assignment

Web Page Scanner with Node.js, TypeScript, and Puppeteer

Table of Contents

Objective

Develop a Node.js application using TypeScript and Puppeteer to scan web pages and **monitor access to sensitive data** on websites. The primary goal is to track any mechanisms that could potentially read or access user input values, form data, or other sensitive information.

Core Mission

Monitor Sensitive Data Access: The target is to detect and document any way that scripts, events, or network requests could access sensitive user data such as input values, form contents, or personal information. As long as you can justify your monitoring approach and explain it well, you have flexibility in the exact implementation schema.

Extract information about web page components, network requests, and page transitions, saving the results in a structured JSON format for security analysis.

🔧 Implementation Flexibility

Schema Adaptability: While the provided structure gives a solid foundation, you have the flexibility to modify or extend this schema as needed for your sensitive data monitoring approach. The key requirements are:

- ✅ **Justify your choices:** Clearly document any schema modifications or additions
- ✅ **Focus on the mission:** Ensure your solution effectively monitors sensitive data access
- ✅ **Monitor input access:** If a script reads an input value, you must report about it - this is the core requirement
- ✅ **Document thoroughly:** Explain your monitoring strategy in your README

Innovation and creative approaches to sensitive data monitoring are encouraged!

Core Requirements

Technology Stack

Node.js TypeScript Puppeteer

Puppeteer Browser Control

- The application should launch a Puppeteer browser instance given a starting URL.
- The browser should remain in "scan mode" until a specific JavaScript function, `finishScan()`, is called from the browser's console.
- The Puppeteer application must expose this `finishScan()` function to the browser's console environment.

Output Storage

- All scan results must be written to a **history** folder.
- A new subfolder should be created within the history folder for each individual scan (e.g., using a timestamp or unique ID as the folder name).
- The final results for each scan should be stored in a JSON file within its respective scan folder.

JSON Output Structure

The JSON output file for each scan should adhere to the following structure:

JSON Structure Example:

```
{
  "parts": [
    {
      "id": "uniquePartId1",
      "type": "html" | "js" | "css" | "image" | "font" | "other",
      "actions": [
        {
          "actionId": "uniqueActionId1",
          "type": "input.value" |
                  "input.addEventListener.change" |
                  "input.addEventListener.input" |
                  "input.addEventListener.keyup" |
                  "input.addEventListener.keydown" |
                  "form.addEventListener.submit" |
                  "httpRequest" |
                  "etc", // Any other mechanisms related to input access or HTTP requests
          "data": "string"
        }
      ]
    }
  ],
  "requests": [
    {
      "actionId": "actionIdThatCausedRequest1",
      "url": "https://example.com/resource.js",
      "requestSource": "script src" | "fetch" | "xhr" |
                       "img src" | "css url" |
                       "link href" | "iframe src",
      "method": "GET" | "POST" | "PUT" | "DELETE" | "HEAD" | "OPTIONS",
      "status": 200
    }
  ],
  "extraPages": [
    {
      "url": "https://example.com/newPage",
      "parts": [
        // Same structure as the top-level "parts" array
      ],
      "requests": [
        // Same structure as the top-level "requests" array
      ]
    }
  ]
}
```

Data Collection Details

parts Array

Each part represents a distinct resource loaded or identified on the page (e.g., HTML document, JavaScript file, CSS file, image).

Fields:

- **id:** A unique identifier for the part within the current scan.
- **type:** Categorization of the resource (e.g., *html*, *js*, *css*, *image*, *font*, *other*).
- **actions:** An array of reports about actions that this part was involved in or caused.

Action Types:

- `input.value`: A direct read of an input element's value property.
- `input.addEventListener.change`: An event listener was attached to an input element for the change event.
- `input.addEventListener.input`: An event listener was attached to an input element for the input event.
- `input.addEventListener.keyup`: An event listener was attached for keyup.
- `input.addEventListener.keydown`: An event listener was attached for keydown.
- `form.addEventListener.submit`: An event listener was attached to a form for the submit event.
- `httpRequest`: An HTTP request was initiated.
- `etc`: Any other mechanisms that could potentially access input values or trigger HTTP requests (e.g., form serialization, form.submit() calls, programmatic form submission, etc.). **Document and justify any additional monitoring approaches you implement.**

Data Field Examples:

- **For input.value:** Just the input identifier (preferably id). Consider cases where inputs don't have IDs - how will you uniquely identify them?
  - With ID: `"id=username"`
  - Without ID: Think about alternative identification strategies
- **For addEventListener:** `"id=username"` or `"name=email"`
- **For form submit:** `"id=loginForm"` or form identifier
- **For httpRequest:** `"https://api.example.com/data"`
- **For etc (custom monitoring):** `"id=contactForm"` or any other justified input access or HTTP-related patterns you identify

requests Array

Each object represents a single HTTP request made during the scan.

- **actionId:** Links back to the actionId from the parts.actions array that caused this request.
- **url:** The full URL of the requested resource.
- **requestSource:** How the request was made (e.g., *script src*, *fetch*, *xhr*, *img src*).
- **method:** HTTP method (e.g., GET, POST).
- **status:** HTTP status code (e.g., 200, 404).

extraPages Array (Optional - Nice to Have)

This feature is optional and not mandatory for the assignment. If implemented, this array captures details for "new pages" encountered during the scan. A "new page" is defined as a significant location change in the browser (e.g., navigation to a new URL, a client-side route change that updates the URL).

Each extraPage object should contain its `url` and then its own `parts` and `requests` arrays, structured identically to the top-level arrays.

Deliverables

- A Git repository (e.g., on GitHub, GitLab) containing the full Node.js project with TypeScript.
- Clear instructions in the **README.md** file on how to set up, build, and run the application.
- Example output JSON files demonstrating the scanner's capabilities.

Evaluation Criteria

- **Primary Goal Achievement:** Effectiveness in monitoring and detecting sensitive data access patterns.
- **Implementation Quality:** Correctness and completeness of the Puppeteer implementation.
- **Schema Adherence & Justification:** Either adherence to the specified JSON output structure, OR well-documented and justified modifications that improve sensitive data monitoring.
- **Data Collection Richness:** Accuracy and comprehensiveness of collected data focusing on input value reads, event listener attachments, and sensitive data access patterns.
- **Action Traceability:** Robustness of linking requests back to the actions that initiated them.
- **TypeScript Implementation:** Proper use of TypeScript for type safety and code organization.
- **Code Quality:** Readability, organization, and maintainability of the codebase.
- **Documentation Excellence:** Clear explanation of approach, schema decisions, and usage instructions in README.md.
- **Innovation Bonus:** Creative approaches to sensitive data monitoring beyond the basic requirements.