# WebSecBot

## Developer Documentation

*Technical Guide for Developers*

WebSecBot Team
2025

A project funded by netidee


https://www.netidee.at/websecbot

# Contents

# 1 Introduction

## 1.1 Project Overview

WebSecBot is a Chrome browser extension designed to help developers identify and fix security vulnerabilities in web applications. The extension combines automated security analysis with an AI-powered assistant to make security testing accessible to developers without extensive security expertise.

This documentation provides technical details for developers who want to understand, modify, or extend the WebSecBot extension.

## 1.2 Technology Stack

WebSecBot is built using the following technologies:

- **Frontend**: React.js, TailwindCSS
- **Browser Integration**: Chrome Extension API
- **Security Analysis**: Custom scanning modules, InternetSecure.org integration
- **AI Backend**: Integration with LLM providers via API (AnythingLLM, Groq, etc.)
- **Data Storage**: Chrome storage API

## 1.3 Repository Structure

The WebSecBot codebase is organized as follows:

```
websecbot/
          public/                # Static assets
          src/                   # Source code
                components/        # React components
                      Generator.js  # Main analysis component
                      Profile.js    # Settings component
                      ui/           # UI components
                utils/             # Utility functions
                      chatGPTUtil.js   # LLM API integration
                      internetSecureScanner.js  # Security scanner
                      localStorage.js  # Storage utilities
                      prompts.js    # LLM prompts
                      routes.js     # Navigation routes
                App.js             # Main application component
                index.js           # Entry point
          public/                # Static assets
                manifest.json      # Chrome extension manifest
          background.js          # Chrome extension background script
```

# 2 Architecture Overview

## 2.1 System Architecture

WebSecBot follows a client-server architecture pattern with the Chrome extension as the client and optional external LLM services as servers.

The main components of the system are:

1. **Chrome Extension**: The frontend interface that integrates with the browser
2. **Background Script**: Handles browser interaction and data collection
3. **Analysis Engine**: Processes collected data to identify security issues
4. **LLM Backend**: Provides AI-powered analysis and conversation capabilities

### 2.2 Data Flow

The data flow in WebSecBot follows these steps:

1. User navigates to a website in Chrome
2. Background script collects webpage data (forms, scripts, headers, etc.)
3. User initiates analysis via the WebSecBot panel
4. Collected data is processed and enhanced with additional scans
5. Data is sent to the LLM backend with security analysis prompts
6. LLM generates comprehensive security analysis
7. Results are displayed to the user in the WebSecBot panel
8. User can interact with the AI assistant for clarification and guidance

## 3 Key Components

### 3.1 Chrome Extension Integration

WebSecBot integrates with Chrome using the extension API. The key integration points are:

#### 3.1.1 Background Script

The background script (`background.js`) runs persistently and handles:

- Initialization of the side panel
- Web request monitoring for header collection
- Content script injection for page data collection (see Section 3.1.2 for details on collected data)
- Message passing between components

```
// Open the side panel when the extension icon is clicked
chrome.action.onClicked.addListener((tab) => {
  chrome.sidePanel.open({ tabId: tab.id });
});

// Configure the side panel to open on extension icon click
chrome.sidePanel.setPanelBehavior({ openPanelOnActionClick: true });

// Listen for tab updates to collect page data
chrome.tabs.onUpdated.addListener(function (tabId, changeInfo, tab) {
  if (changeInfo.status === "complete" && tab.active) {
```

```
    if (!tab.url?.startsWith("chrome://")) {
      chrome.scripting.executeScript({
        target: { tabId },
        func: loadWebPageInfo,
      })
      // ...
    }
  }
});
```

Listing 1: Key Background Script Functions

### 3.1.2 Content Scripts

Content scripts are injected into the webpage to collect security-relevant data:

- Forms and input fields
- Scripts and JavaScript patterns
- Cookies and meta tags
- HTML comments
- Directory structure testing

```
function loadWebPageInfo() {
  const webInfo = {
    webUrl: window.location.href,
    title: document.title,
    forms: [],
    inputs: [],
    scripts: [],
    metaTags: [],
    cookies: document.cookie,
    // ...
  };

  // Extract forms
  document.querySelectorAll('form').forEach((form) => {
    webInfo.forms.push({
      action: form.getAttribute('action') || 'N/A',
      method: form.getAttribute('method') || 'GET',
      fields: Array.from(form.querySelectorAll('input, textarea')).map((
   field) => ({
        name: field.getAttribute('name') || 'N/A',
        type: field.getAttribute('type') || 'N/A',
        value: field.getAttribute('value') || 'N/A',
      })),
    });
  });

  // Extract other data...

  return webInfo;
}
```

Listing 2: Web Page Data Collection

## 3.2 Security Analysis Engine

The security analysis is performed through a combination of:

1. **Client-side Data Collection**: Gathering data from the current page
2. **External Security Scanning**: Using InternetSecure.org
3. **LLM-based Analysis**: Processing the collected data with AI

### 3.2.1 InternetSecure Integration

WebSecBot uses the InternetSecure.org service to perform additional security checks:

```
export async function scanWithInternetSecure(url) {
  try {
    const host = new URL(url).hostname;
    const scanUrl = `https://internetsecure.org/?domain=${
   encodeURIComponent(host)}`;

    console.log(`Scanning ${host} with InternetSecure.org...`);
    const { data: html } = await axios.get(scanUrl, {
      headers: { "User-Agent": "Mozilla/5.0" },
      timeout: 60000
    });

    // Load the HTML into Cheerio
    const $ = cheerio.load(html);
    const sections = {};

    // Extract sections from the scan results
    $("h3").each((_, h3) => {
      const title = $(h3).text().replace(/:$/, "").trim();
      const body = $(h3).nextUntil("h3").text().trim();
      if (title && body) {
        sections[title] = body;
      }
    });

    return { rawHtml: html, sections };
  } catch (err) {
    console.error("Error scanning with InternetSecure:", err);
    return { error: err.message };
  }
}
```

Listing 3: Internet Secure Scanner Integration

### 3.2.2 Privacy Considerations for External Scanning

WebSecBot sends domain information to InternetSecure.org for certain security checks. The following data privacy considerations apply:

- Only the hostname (domain) is sent to InternetSecure.org, not full URLs or page content
- No user personal data is shared with external services
- User IP addresses may be visible to InternetSecure.org when requests are made

- No user account or identifying information is required for these scans

All scanning is performed on-demand and only when explicitly requested by the user. The extension's privacy policy should be reviewed for complete details on data handling.

### 3.2.3 Directory Security Scanning

WebSecBot includes functionality to check for sensitive directories and files:

```
async function checkFtpDirectories(origin) {
  // Comprehensive path list ordered by likelihood of discovery
  const directoryPaths = [
    '/ftp',
    '/admin',
    '/uploads',
    '/files',
    '/backup',
    // ...additional paths...
  ];

  try {
    // Get current active tab
    const tabs = await chrome.tabs.query({active: true, currentWindow:
   true});
    if (!tabs || tabs.length === 0) {
      return [];
    }

    // Execute the scan in the context of the current page
    const scanResults = await chrome.scripting.executeScript({
      target: { tabId: tabs[0].id },
      func: scanFtpPathsInPageContext,
      args: [directoryPaths, origin]
    });

    if (scanResults && scanResults[0] && scanResults[0].result) {
      return scanResults[0].result;
    }
    return [];
  } catch (error) {
    console.error("Error in security directory scanning:", error);
    return [];
  }
}
```

Listing 4: Directory Security Scanning

## 3.3 LLM Integration

WebSecBot integrates with Large Language Models (LLMs) to provide AI-powered security analysis and assistance.

### 3.3.1 LLM API Integration

The extension uses an API client to communicate with different LLM providers:

```
export const postChatGPTMessage = async (message, conversationHistory,
    setConversationHistory, openAIKey, model = CHATGPT_MODEL, apiUrl =
    CHATGPT_END_POINT) => {
  // Set headers for the axios request
  const config = {
    headers: {
      Authorization: `Bearer ${openAIKey}`,
    },
  };

  // Create the message object to send to the API
  var messages = conversationHistory;
  const userMessage = { role: "user", content: message };
  messages.push(userMessage);

  // Define the data to send in the request body
  const chatGPTData = {
    model: model,
    messages: messages,
  };

  try {
    // Send a POST request to the LLM API
    const response = await axios.post(apiUrl, chatGPTData, config);

    // Extract the message content from the API response
    var message = response?.data?.choices[0]?.message;

    //add the response message to the history
    messages.push(message);
    setConversationHistory(messages);

    // Return the message content
    return message.content;
  } catch (error) {
    console.error("Error with ChatGPT API");
    console.error(error);
    return null;
  }
};
```

Listing 5: LLM API Integration

### 3.3.2 Security Analysis Prompts

The extension uses carefully designed prompts to guide the LLM in performing security analysis:

```
export const COMPREHENSIVE_SECURITY_ANALYSIS = `
You are a security expert and web security assistant focused on OWASP
    Top 10 vulnerabilities. Perform a comprehensive security analysis of
    the following web page covering major OWASP Top Ten security
    categories.

## Overall Instructions:
1. Analyze the provided web page data for security vulnerabilities
    across OWASP Top Ten security categories.
2. First, provide a brief executive summary of the website's security
    posture (3-4 sentences).
```

```
3. Include ONLY findings that have solid EVIDENCE in the provided data.
4. Focus ONLY on client-side observable issues that can be detected from
    the provided data.
5. For each category:
   - Explain why the category is important according to OWASP
   - List specific findings with evidence
   - Provide criticality levels and explanations
   - Offer actionable remediation steps

## Security Categories to Analyze:
### 1. Broken Access Control (BAC)
### 2. Injection Vulnerabilities
### 3. Cryptographic Failures
### 4. Security Misconfiguration
### 5. Vulnerable & Outdated Components

## Input Details:
Web URL: {webUrl}
Forms: {forms}
Inputs: {inputs}
Cookies: {cookies}
Scripts: {scripts}
JS Elements: {jsElements}
Meta Tags: {metaTags}
Comments: {comments}
Headers: {headers}
FTP Directories: {ftpDirectories}
Mixed Content: {mixedContent}

## Security Scan Results:
{securityScan}

## Output Format:
[Detailed format instructions follow...]
`;
```

Listing 6: Example Security Analysis Prompt (Simplified)

## 3.4 User Interface Components

The WebSecBot UI is built with React and consists of several key components:

### 3.4.1 Main UI Structure

The extension uses a tabbed interface with two main sections:

- **Analysis Tab**: For running and viewing security scans
- **Security Chat Tab**: For interacting with the AI assistant

```
{/* Tab navigation */}
<div className="border-b border-gray-200 bg-gray-50">
  <div className="px-4 pt-4">
    <div className="grid w-full grid-cols-2 h-10 bg-gray-100 p-1 text-
   gray-500">
      <button
```

```
        className={`inline-flex items-center justify-center whitespace-
    nowrap px-3 py-1.5 text-sm font-medium ${
          activeTab === "analysis"
            ? "bg-white text-[#1e4da1] shadow-sm"
            : "hover:bg-gray-200"
        }`}
        onClick={() => setActiveTab("analysis")}
      >
        Analysis
      </button>
      <button
        className={`inline-flex items-center justify-center whitespace-
    nowrap px-3 py-1.5 text-sm font-medium ${
          activeTab === "security-chat"
            ? "bg-white text-[#1e4da1] shadow-sm"
            : "hover:bg-gray-200"
        }`}
        onClick={() => setActiveTab("security-chat")}
      >
        Security Chat
      </button>
    </div>
  </div>

  {/* Conditional rendering of tab content */}
  {activeTab === "analysis" && (
    // Analysis tab content
  )}

  {activeTab === "security-chat" && (
    // Chat tab content
  )}
</div>
```

Listing 7: UI Tab Navigation

### 3.4.2 Analysis Component

The Analysis component handles the security scanning functionality:

- Initiating security scans
- Processing and formatting scan results
- Displaying the formatted analysis

```
const handleComprehensiveAnalysis = async () => {
  setIsLoading(true);
  setHasAnalyzed(false);

  // Update the analysis content state
  setAnalysisContent("Initializing comprehensive security analysis...");

  try {
    // Load data from chrome.storage
    const webUrl = await loadData('webUrl');
    const webPageInfo = await loadData('webPageInfo');
```

```
    if (!webPageInfo) {
      // Handle missing data error
      return;
    }

    // Convert data to text for the LLM
    const formsText = stringifyForms(webPageInfo.forms);
    const inputsText = stringifyInputs(webPageInfo.inputs);
    // More data conversions...

    // Run security scan
    const scanResults = await scanWithInternetSecure(url);
    securityScanText = formatSecurityScanForLLM(scanResults);

    // Prepare analysis fields
    const analysisFields = {
      webUrl: webPageInfo.webUrl || webUrl || "N/A",
      forms: formsText,
      // More fields...
      securityScan: securityScanText,
    };

    // Execute the comprehensive analysis prompt
    const response = await executeCommand(prompts.
   COMPREHENSIVE_SECURITY_ANALYSIS, analysisFields, []);

    setAnalysisContent(response);
    setHasAnalyzed(true);

    // Save the analysis content to storage
    saveData('savedAnalysisContent', response);
    saveData('analysisWebUrl', webPageInfo.webUrl || webUrl || "N/A");

  } catch (error) {
    console.error(error);
    setAnalysisContent("Error: " + error.message);
  } finally {
    setIsLoading(false);
  }
};
```

Listing 8: Analysis Component - Key Function

### 3.4.3 Chat Component

The Chat component provides the interactive AI assistant functionality:

- Sending user questions to the LLM
- Displaying AI responses with markdown formatting
- Context toggling to include/exclude analysis results

```
const handleSendGeneralQuestion = async () => {
  if (!generalQuestion.trim() || isGeneralSending) {
    return;
```

```
  }

  setIsGeneralSending(true);
  const newQuestion = generalQuestion;
  setGeneralQuestion('');

  // Format the question for display
  const formattedQuestion = `## Your Question\n${newQuestion}\n\n`;
  setChatContent(formattedQuestion + "Thinking...");

  try {
    // Add system message for security context
    const securitySystemPrompt = `You are a web security expert
   assistant...`;

    // Check if we should include analysis context
    let fullPrompt;

    if (includeAnalysisContext && hasAnalyzed && analysisContent) {
      // Include the analysis results as context
      const analysisWebUrl = await loadData('analysisWebUrl') || "
   unknown website";
      fullPrompt = `${securitySystemPrompt}\n\n
WEBSITE SECURITY ANALYSIS CONTEXT:
${analysisContent}

USER QUESTION: ${newQuestion}`;
    } else {
      // Regular question without analysis context
      fullPrompt = `${securitySystemPrompt}\n\nUser question: ${
   newQuestion}`;
    }

    // Send to LLM
    const response = await postChatGPTMessage(
      fullPrompt,
      [],
      () => {},
      openAIKey,
      model,
      apiUrl
    );

    // Update UI with response
    setChatContent(formattedQuestion + `## Expert Response\n${response
   }`);

  } catch (error) {
    console.error("Chat error:", error);
    setChatContent(formattedQuestion + "Error: " + error.message);
  } finally {
    setIsGeneralSending(false);
  }
};
```

Listing 9: Chat Component - Key Function

### 3.4.4 Settings Component

The Profile/Settings component allows users to configure their LLM backend:

```
function Profile({ setPage, openAIKey, setOpenAIKey, model, setModel,
    apiUrl, setAPIUrl }) {
  const handleSubmit = (e) => {
    e.preventDefault();
    const formData = new FormData(e.target);
    const updatedOpenAIKey = formData.get("openAIKey");
    const updatedModel = formData.get("model");
    const updatedApiUrl = formData.get("apiUrl");

    // Update state and save to storage
    setOpenAIKey(updatedOpenAIKey);
    saveData('openAIKey', updatedOpenAIKey);
    setModel(updatedModel);
    saveData('model', updatedModel);
    setAPIUrl(updatedApiUrl);
    saveData('apiUrl', updatedApiUrl);
  }

  return (
    // Form UI for settings
  );
}
```

Listing 10: Settings Component

# 4 Development Guide

## 4.1 Setting Up the Development Environment

To set up a development environment for WebSecBot:

1. Clone the repository from https://github.com/sschritt/WebSecBot
2. Install dependencies:

   ```
   npm install
   ```

3. Build the extension:

   ```
   npm run build
   ```

4. Load the extension in Chrome:
   - Navigate to `chrome://extensions/`
   - Enable "Developer mode"
   - Click "Load unpacked" and select the `build` directory

## 4.2 Connecting to LLM Providers

WebSecBot can connect to various LLM providers through an OpenAI-compatible API interface. The recommended setup uses AnythingLLM as a backend, which provides flexibility to use either external API providers or run models locally.

### 4.2.1 AnythingLLM Configuration

To use AnythingLLM as the LLM backend:

1. Install AnythingLLM following the instructions at https://github.com/Mintplex-Labs/anything-llm
2. Configure a workspace in AnythingLLM to use with WebSecBot
3. Note the workspace name as it will be used as the model name in WebSecBot
4. In WebSecBot's settings:
   - API URL: Enter the AnythingLLM endpoint (e.g., `http://localhost:3001/api/v1/openai/chat/`
   - Model: Enter the exact name of your AnythingLLM workspace
   - API Key: Enter "sk-1234" (default) or your custom API key if configured

### 4.2.2 External API Providers

WebSecBot can also connect directly to OpenAI-compatible API providers:

1. Obtain API credentials from your chosen provider (e.g., OpenAI, Groq)
2. In WebSecBot's settings:
   - API URL: Enter the provider's endpoint (e.g., `https://api.openai.com/v1/chat/completions`)
   - Model: Enter the model name (e.g., `gpt-4`)
   - API Key: Enter your API key from the provider

**Development Note:** The code uses a standard Bearer token authentication format which works with OpenAI-compatible APIs. When using AnythingLLM, the token is still sent in the same format, but AnythingLLM may handle authentication differently on the server side.

### 4.2.3 Default LLM Configuration Values

WebSecBot comes with pre-configured default values for connecting to AnythingLLM in the `src/utils/chatGPTUtil.js` file:

```
// Define constants
const CHATGPT_END_POINT = "http://localhost:3001/api/v1/openai/chat/
    completions";
const CHATGPT_MODEL = "WebSecBot";
```

Listing 11: Default LLM Connection Constants

These values are used when no custom settings are provided. Developers can modify these defaults directly in the source code to match their preferred LLM setup. For production deployments, these should be customized to point to a reliable LLM backend.

## 4.3 Development Workflow

The recommended development workflow is:

1. Make changes to the source code
2. Rebuild the extension:

```
npm run build
```

3. Refresh the extension in Chrome:
   - Go to `chrome://extensions/`
   - Click the refresh icon on the WebSecBot card
4. Test your changes by using the extension

## 4.4 Adding New Security Checks

To add new security checks to WebSecBot:

1. Identify the type of security check you want to add
2. Modify the appropriate data collection function (e.g., `loadWebPageInfo` for client-side checks)
3. Update the security analysis prompt in `prompts.js` to include the new check
4. Test the new check on websites with known vulnerabilities

Example of adding a new security check for detecting outdated libraries:

```
// In loadWebPageInfo()
// Add detection for library versions
document.querySelectorAll('script').forEach((script) => {
  const src = script.getAttribute('src') || '';
  const content = script.innerText;

  // Look for jQuery version
  const jQueryVersionMatch = src.match(/jquery-(\d+\.\d+\.\d+)\.js/i) ||
                             content.match(/jQuery v(\d+\.\d+\.\d+)/i);
  if (jQueryVersionMatch) {
    if (!webInfo.libraryVersions) webInfo.libraryVersions = [];
    webInfo.libraryVersions.push({
      library: 'jQuery',
      version: jQueryVersionMatch[1]
    });
  }

  // Add more library detection as needed
});

// Then update the prompt to include library version analysis
// In prompts.js:
export const COMPREHENSIVE_SECURITY_ANALYSIS = `
...
## Input Details:
...
```

```
Library Versions: {libraryVersions}
...

### 5. Vulnerable & Outdated Components
Check for evidence of:
- Known vulnerable versions of JavaScript libraries (jQuery < 3.5.0, etc
    .)
...
`;
```

Listing 12: Adding a New Security Check

### 4.5 Modifying the LLM Prompts

The LLM prompts in `prompts.js` define how the security analysis is performed. To modify them:

1. Understand the current prompt structure and its purpose
2. Make focused changes to improve specific aspects
3. Test the changes with a variety of websites
4. Refine based on the results

Guidelines for effective prompts:

- Be specific about what you want the LLM to analyze
- Clearly define the input data format
- Specify the desired output structure
- Include examples for complex tasks
- Use consistent terminology
- Avoid ambiguous instructions

## 5 Testing

### 5.1 Manual Testing

Manual testing is essential for validating the extension's functionality:

1. **Installation Testing**: Verify the extension installs correctly and appears in Chrome
2. **UI Testing**: Test all user interface elements and interactions
3. **Scan Testing**: Test the analysis functionality on various websites
4. **Chat Testing**: Verify the chat functionality with different types of questions
5. **Settings Testing**: Test saving and loading different configuration options

### 5.2 Test Websites

The following websites are useful for testing specific vulnerabilities:

- **OWASP Juice Shop**: https://juice-shop.herokuapp.com/
- **DVWA (Damn Vulnerable Web Application)**: Requires local setup
- **WebGoat**: https://github.com/WebGoat/WebGoat
- **Vulnweb**: http://testphp.vulnweb.com/
- **Hackazon**: https://github.com/rapid7/hackazon

### 5.3 Common Issues and Debugging

When debugging WebSecBot, look for these common issues:

1. **Chrome Extension Permissions**: Make sure the extension has the necessary permissions
2. **LLM API Rate Limits**: API providers may have rate limits or token limits
3. **Cross-Origin Restrictions**: Security checks may be blocked by CORS

Use Chrome's developer tools to debug:

- Open the Chrome DevTools (F12)
- Go to the "Extensions" tab (or navigate to chrome://extensions, click "background page" under WebSecBot)
- Check the console for errors and logs

## 6 Deployment

### 6.1 Building for Production

To build the extension for production:

```
npm run build
```

This creates a production-ready build in the `build` directory.

### 6.2 Packaging the Extension

To package the extension for distribution:

1. Create a ZIP file of the `build` directory
2. Ensure the manifest.json has the correct version number
3. Include any necessary documentation

## 7 Extending WebSecBot

### 7.1 Community Contributions

Guidelines for community contributions:

- Follow the existing code structure and style
- Write clear commit messages
- Include documentation for new features
- Add tests for new functionality
- Submit pull requests with a clear description of changes

# 8 Appendix

## 8.1 API Reference

### 8.1.1 Chrome Extension API

Key Chrome Extension APIs used in WebSecBot:

- `chrome.sidePanel`: For creating and managing the side panel
- `chrome.tabs`: For interacting with browser tabs
- `chrome.scripting`: For injecting content scripts
- `chrome.storage`: For storing extension data
- `chrome.webRequest`: For monitoring web requests

## 8.2 Glossary

**OWASP** The Open Web Application Security Project, a nonprofit foundation that works to improve software security.

**LLM** Large Language Model, an AI system capable of understanding and generating human-like text.

**XSS** Cross-Site Scripting, a security vulnerability that allows attackers to inject client-side scripts into web pages.

**CORS** Cross-Origin Resource Sharing, a security feature that restricts web pages from making requests to a different domain.

**CSP** Content Security Policy, a security layer that helps detect and mitigate certain types of attacks.

**RAG** Retrieval-Augmented Generation, a technique that enhances LLM outputs by retrieving relevant information from a knowledge base.

## 8.3 Resources

- Chrome Extension Documentation: https://developer.chrome.com/docs/extensions/
- OWASP Top Ten: https://owasp.org/www-project-top-ten/
- React Documentation: https://reactjs.org/docs/getting-started.html
- AnythingLLM Documentation: https://github.com/Mintplex-Labs/anything-llm