International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I12P121

Volume 72 Issue 12, 171-178, December 2024


*Review Article*

# WebView Security Best Practices

Sheshananda Reddy Kandula

*Adobe Inc., NJ, USA.*

*Corresponding Author : Sheshananda4u@gmail.com*

Received: 07 November 2024 Revised: 01 December 2024 Accepted: 19 December 2024 Published: 31 December 2024

***Abstract*** *- WebViews play a great role in mobile and desktop applications by embedding web content within native applications. Native applications, typically written in platform-specific languages and frameworks, often share a common backend with multiple web-based clients, like Android, iOS, Windows, and macOS. To streamline development processes and enhance cross-platform compatibility, developers leverage WebViews as a unifying component. While WebViews offer substantial advantages in terms of development speed, flexibility, and code reuse, they inherently introduce security vulnerabilities if not implemented securely. Significant research has been performed on vulnerabilities in WebViews in different platforms [1], [2], [3], [4], but there is a lack of a consolidated repository of best practices for securely implementing WebViews. This review aims to address the gap and systematically investigates prevalent WebView security vulnerabilities, assess their potential impact on application security and user privacy, and provide best practices. By bridging the gap in existing literature, this work provides developers with actionable guidelines to build more resilient and secure WebViews usage in mobile and desktop environments.*

***Keywords*** *- Android apps, Electron apps, iOS apps, Security, WebView.*

## 1. Introduction

WebViews are widely utilized components in software development that enable mobile and desktop applications to render web content within a native environment. By embedding a web browser directly into applications, WebViews facilitate the seamless integration of web-based features and dynamic content, making them indispensable in hybrid application development. They are particularly prevalent in mobile platforms such as Android (via WebView) and iOS (via WKWebView), as well as desktop frameworks like Electron and cross-platform development solutions such as Flutter and React Native. Through these implementations, WebViews act as a bridge between web technologies and native platforms, enabling developers to create interactive and feature-rich applications while leveraging the versatility of the web.

However, alongside their many advantages, WebViews introduce significant security challenges. By inherently relying on web-based technologies, they inherit vulnerabilities traditionally associated with web browsers, such as cross-site scripting (XSS) [5], [6] remote code execution (RCE) [7], man-in-the-middle (MITM) [8] attacks, and data leakage. These risks are exacerbated in scenarios where untrusted content is loaded or where WebView configurations are mismanaged. Such vulnerabilities not only compromise application security but can also expose sensitive user data and other critical assets to malicious actors. Addressing these challenges requires a comprehensive understanding of WebView security risks and the adoption of robust mitigation strategies. This introduction highlights the critical need for developers to balance the convenience of Web Views with robust security practices. Proper configuration, secure coding principles, and adherence to best practices are essential for mitigating risks and safeguarding applications and users. The following sections explore the features of Web Views, their vulnerabilities, and the strategies that developers can employ to enhance their security.

## 2. WebView Definition

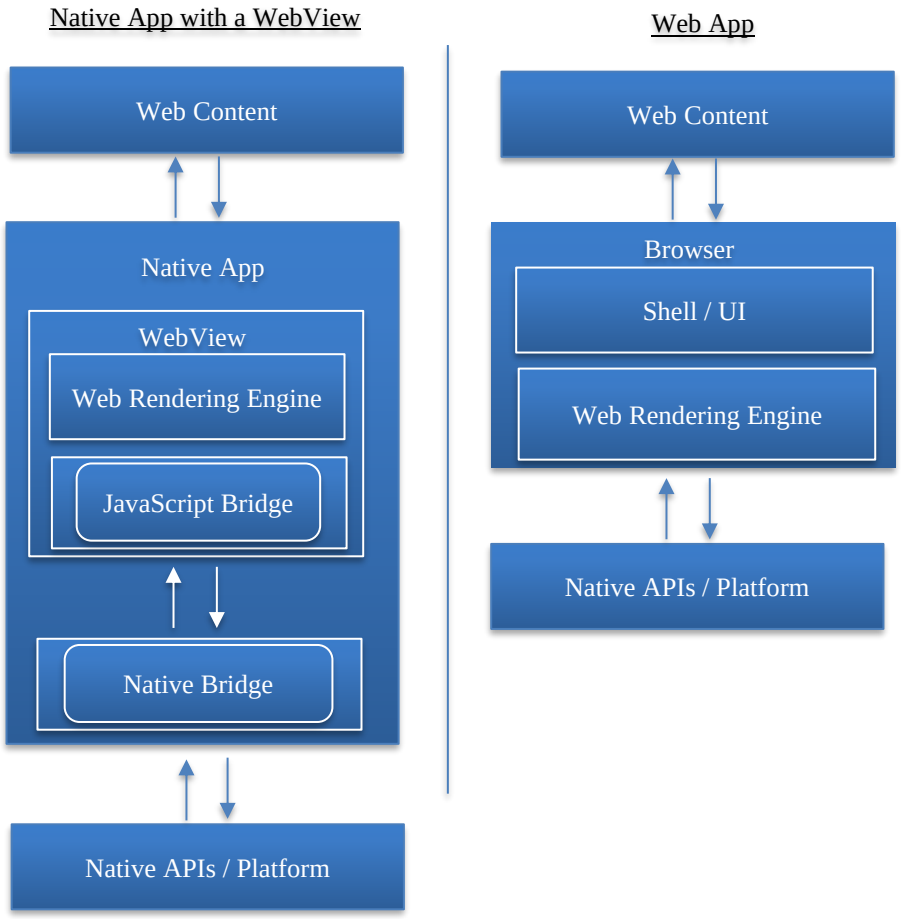
Android WebView[9]: The WebView class is an extension of Android's View class that displays web pages. WebView will not have the features of a fully developed web browser, such as navigation controls or an address bar. All WebView does, by default, is show a web page. iOS WebView[10]: A WKWebView object is a platform-native view to incorporate web content seamlessly into the app's UI. A web view supports a full web-browsing experience and presents HTML, CSS, and JavaScript content alongside the app's native views.

Even though there are several platform-specific implementations and terminologies for WebViews—such as Android WebView, iOS WKWebView, and Electron WebView—this review adopts a unified perspective.

Going forward, "WebView" will be used as a general term to represent any platform-specific implementation of a web content rendering engine embedded within native applications.

### *2.1. Native App with a WebView vs. Web App*

Native App running on Mobile devices or Desktop use WebView to display the web content[11]. Browsers will have a full rendering engine to run the web content[12].




## 3. Features of WebViews

The following features are designed to make WebViews versatile and powerful for integrating web technologies into native and hybrid applications.

- Cross-Platform Content Rendering: Embeds web content (HTML, CSS, JavaScript) in native apps for hybrid app development.
- JavaScript Execution: Supports running JavaScript for dynamic functionality, with options to enable/disable on Android, iOS, and Electron.
- Bidirectional Communication: Enables interaction between native code and web content via bridges like addJavascriptInterface (Android), WKScriptMessageHandler (iOS), and ipcRenderer (Electron).
- Navigation Controls: Allows control over URL loading and restricts content to trusted domains.
- Content Security: Implements Content Security Policies (CSP), SSL validation, and safe browsing modes.
- File and Resource Access: Limits file system access using platform-specific permissions and sandboxing.
- Offline Caching: Supports caching for loading web content offline.
- Customization and Styling: Offers UI customization, including injecting CSS/JavaScript for tailored user experiences.
- Debugging and Performance Tools: Integrated debugging via Chrome DevTools (Android/Electron) and Safari Developer Tools (iOS).
- Accessibility: Inherits platform-specific accessibility APIs for screen readers and navigation.
- Advanced Features: Platform-specific tools like startSafeBrowsing (Android), desktop/mobile rendering modes (iOS), and Node.js integration (Electron).

## 4. How WebView Vulnerabilities occurs

WebView vulnerabilities exist primarily because of the inherent nature of WebViews, which combine web technologies with native application features.

This creates a complex surface where web-based security risks (e.g., XSS, malicious code injection) interact with the security implications of native app environments (e.g., accessing device APIs, files, or secure data).

### *4.1. XSS (Cross Site Scripting)[5]*

Cross Site Scripting (XSS)[13], [14] is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application by executing malicious JavaScript code in the victim's browser.

### *4.2. RCE (Remote Code Execution)*

Remote Code Execution (RCE) is a kind of security vulnerability that allows attackers to execute arbitrary code in the victim's machine. It is a technique for accessing or infiltrating the data in a target machine or system. [15], [16]

### *4.3. Code Injection*

Code injection attack occurs when untrusted code is injected into the application, and the application executes the code without proper validations. [17]

The following are a few reasons why these vulnerabilities occur:

- Trusting the user's input
- Misconfigurations
- MITM attacks
- File System Access

## 5. Overview of Related Work

The studies referenced below touch on various dimensions of WebView security. However, there is a lack of comprehensive best practices for securing WebViews in real-world applications. Here is a summary of the contributions:

### *5.1. Web Access Monitoring Mechanism via Android WebView for Threat Analysis [18]*

- This paper focuses on using Android WebView to monitor web access patterns and detect threats.
- Contribution: Offers insights into how WebView can serve as a tool for threat analysis.
- Gap: Does not address practical, preventive measures for securing WebView itself.

### *5.2. The Privacy and Security Risks of Mobile In-App Browsers [19]*

- This study highlights the privacy and security risks posed by in-app browsers and WebViews.
- Contribution: Identifies risks such as tracking, phishing, and lack of transparency in mobile WebViews.
- Gap: Provides limited actionable strategies for developers to mitigate these risks.

### *5.3. Automatically Retrofitting Cordova Applications for Stricter Content Security Policies [4]*

- Examines how Cordova applications can implement stricter Content Security Policies (CSPs).
- Contribution: Focuses on retrofitting legacy apps with enhanced CSP measures.
- Gap: Discusses Cordova apps but lacks a generalized framework for securing WebViews across platforms.

### *5.4. Comparative Analysis of Android and iOS from Security Viewpoint [3]*

- Compares Android and iOS platforms regarding security features and vulnerabilities.
- Contribution: Provides a platform-level perspective on security.
- Gap: Does not focus specifically on WebView or hybrid app security.

### *5.5. Vulnerabilities in Android WebView objects: Still not the end! [20]*

### *5.6. Electrolint and Security of Electron Applications [21]*

## 6. Common WebView Vulnerabilities

Below are common vulnerabilities observed so far in the WebViews from the research. Even Android, iOS and Electron recommend using best practices against these vulnerabilities.

### *6.1. Real World Vulnerabilities*

Several applications were vulnerable to WebView implementations.

- Basecamp WebView validation [22]: It was identified that the android com.basecamp.bc3 application contains a Webview where the loaded URLs are not sanitised properly. As this web view's functionality is extended via javascript interfaces and has the javascript enabled, it is possible to inject arbitrary javascript code, which will be executed by the application's webview and provide access to the java native code.

- Zomato exported activity WebView Vulnerability [23]

- TikTok One Click Account Hijacking via Unvalidated Deeplink [24]: A WebView Hijacking vulnerability was found on the TikTok Android application via an unvalidated deep link on an un-sanitized parameter. This could have resulted in account hijacking through a JavaScript interface.

**Table 1. WebView vulnerabilities**

| Vulnerability | Description |
|---|---|
| **JavaScript Injection** | Malicious JavaScript code is injected into WebView, allowing attackers to execute arbitrary code within the application, potentially leading to data theft and manipulation of app functionality. |
| **File System Access** | WebViews may inadvertently grant access to a device's file system, potentially exposing sensitive data or allowing malicious file operations like reading, writing, or deleting files. |
| **SSL Certificate Validation** | Inadequate SSL certificate validation can leave WebViews vulnerable to **Man-in-the-Middle (MITM)** attacks, allowing attackers to intercept and manipulate network traffic. |
| **Cross-Site Scripting (XSS)** | WebViews are prone to **XSS** attacks when they render untrusted content without proper sanitization, allowing attackers to inject malicious scripts and compromise user data or functionality. |
| **URL Scheme Abuse** | Custom URL schemes supported by WebViews can be exploited by attackers to trigger unintended actions within the app or access sensitive information, potentially leading to vulnerabilities. |
| **Insecure Bridge Communication** | The communication bridge between native code and web content can be a target for attacks if not properly secured, enabling attackers to execute malicious code or access sensitive native APIs. |

## 7. Security Best Practices

These best practices provide a strong foundation for securing WebViews in mobile and desktop applications, enhancing overall app security and protecting user data.

### *7.1. Input Validation and Sanitization*

Description**:** Ensure all user input and external data are validated and sanitized to prevent injection attacks.

### *7.2. Content Security Policy (CSP)*

Description: Implement a strict Content Security Policy (CSP) to restrict the sources from which executable content can be loaded, mitigating XSS and injection risks.

### *7.3. SSL Pinning*

Description: Apply SSL pinning to ensure WebViews only connect to trusted servers with valid certificates, preventing Man-in-the-Middle (MITM) attacks.

### *7.4. Disable JavaScript When Unnecessary*

Description: Disable JavaScript unless absolutely required, reducing the attack surface for potential exploits.

### *7.5. Limit File System Access*

Description: Restrict WebView's access to the device's file system to prevent malicious file operations and unauthorized data exposure.

### *7.6. Secure Bridge Communication*

Description: Encrypt and authenticate data exchanged between native code and web content, ensuring secure communication channels.

### *7.7. Regular Security Audits*

Description: Conduct regular security audits to identify and address vulnerabilities and apply security patches promptly.

**Table 2. WebView Security Best Practices**

| Category | **Android**[25] | **iOS** [26] | **Electron** |
|---|---|---|---|
| Input Validation and Sanitization | Validate and sanitize all user inputs and external data before rendering in WebViews to prevent injection attacks. | Same as Android | Same as Android |
| Content Security Policy (CSP) | Implement CSP in loaded web content to restrict sources of executable scripts and mitigate XSS. | Apply CSP in WKWebView's content using the HTTP header or meta tags. | Set CSP for Electron apps to restrict sources of scripts and other content (default-src 'self'). |
| JavaScript Management | Disable JavaScript unless necessary using *getSettings(). setJavaScriptEnabled(false).* | Disable JavaScript unless required using *configuration.preferences. javaScriptEnabled = false.* | Set javascript: false in webPreferences unless necessary. |

| | | | |
|---|---|---|---|
| File System Access | Restrict access to local files with file:// URLs and disable file access via setAllowFileAccess(false). | Same as Android; WKWebView automatically has limited file system access. | Use sandbox: true and limit file system access through webPreferences. |
| Secure Bridge Communication | Use addJavascriptInterface cautiously, exposing only necessary interfaces and securing them with access controls. | Limit message handlers in WKWebView using WKScriptMessageHandler to prevent exploitation. | Secure communication bridges and encrypt sensitive data exchanged between web content and native code. |
| URL Allowlisting | Implement URL filtering using shouldOverrideUrlLoading() to allow only trusted URLs. | Use WKNavigationDelegate to validate and restrict navigations to trusted domains. | Use will-navigate and webContents listeners to restrict WebView navigation. |
| Disable Debugging Features | Disable debugging with setWebContentsDebuggingEnabled (false) for production builds. | Disable Web Inspector in production by setting configuration. Preferences.isInspectable = false. | Disable enableRemoteModule and developer tools in production builds. |
| Prevent XSS and Injection Attacks | Use WebViewClient with input sanitization CSP and disable untrusted JavaScript execution. | Same as Android | Apply CSP, disable untrusted JavaScript, and sanitize inputs. |
| Clear Cache and Cookies | Use clear cache () and CookieManager to remove cached data and cookies after each session. | Use websiteDataStore.removeData to clear sensitive data periodically. | Clear cache and cookies regularly in WebView to prevent sensitive data leakage. |
| Node.js and Native Code Access | NA | NA | Disable Node.js integration with nodeIntegration: false and use contextIsolation: true. |

## 8. Conclusion

WebViews are powerful components of mobile and web applications, but they also introduce potential security risks. By understanding common vulnerabilities and implementing robust security measures, developers can significantly enhance the security of WebView implementation. Regular security assessments, adherence to best practices, and staying informed about emerging threats are crucial for maintaining the integrity and security of applications utilizing WebViews.

## Funding Statement

This research was conducted as a part of my self-interest, and this research was self-funded.

## References

[1] Zihao Jin et al., "A Security Study about Electron Applications and a Programming Methodology to Tame DOM Functionalities," *Network and Distributed System Security Symposium*, San Diego, CA, USA, pp. 1-16, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[2] Chi-Yu Li et al., "Privacy Leakage and Protection of Input Connection Interface in Android," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3309-3323, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[3] Shivi Garg, and Niyati Baliyan, "Comparative Analysis of Android and iOS from Security Viewpoint," *Computer Science Review*, vol. 40, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[4] Basil Schöni, "*Automatically Retrofitting Cordova Applications for Stricter Content Security Policies*," Bachelor Thesis, University of Bern, pp. 1-84, 2020. [Google Scholar] [Publisher Link]

[5] KirstenS, Cross Site Scripting (XSS), OWASP Foundation, 2025. [Online]. Available: https://owasp.org/www-community/attacks/xss/

[6] CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') (4.16), Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/data/definitions/79.html

[7] CWE-749: Exposed Dangerous Method or Function (4.16), Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/data/definitions/749.html

[8] CWE-300: Channel Accessible by Non-Endpoint (4.16), Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/data/definitions/300.html

[9] Build Web Apps in WebView, Android Developers. [Online]. Available: https://developer.android.com/develop/ui/views/layout/webapps/webview
[10] WKWebView, Apple Developer Documentation. [Online]. Available: https://developer.apple.com/documentation/webkit/wkwebview
[11] kirupa Chinnathambi, Understanding WebViews, Kirupa, 2025. [Online]. Available: https://www.kirupa.com/apps/webview.htm
[12] Populating the Page: How Browsers Work - Web Performance, MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work
[13] Andrew R. Regenscheid, and Geoff Beier, "*Security Best Practices for the Electronic Transmission of Election Materials for UOCAVA Voters*," National Institute of Standards and Technology, Internal Report, pp. 1-73, 2011. [CrossRef] [Google Scholar] [Publisher Link]
[14] Paul A. Grassi, Michael E. Garcia, and James L. Fenton, "*Draft NIST Special Publication 800-63-3 Digital Identity Guidelines*," National Institute of Standards and Technology, pp. 1-34, 2017. [Google Scholar] [Publisher Link]
[15] Feng Xiao et al., "Understanding and Mitigating Remote Code Execution Vulnerabilities in Cross-Platform Ecosystem," *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Los Angeles CA USA, pp. 2975-2988, 2022. [CrossRef] [Google Scholar] [Publisher Link]
[16] Mir Masood Ali et al., "Rise of Inspectron: Automated Black-box Auditing of Cross-Platform Electron Apps," *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 1-18, 2024. [Google Scholar] [Publisher Link]
[17] Weilin Zhong, and Rezos, Code Injection, OWASP. [Online]. Available: https://owasp.org/www-community/attacks/Code_Injection
[18] Yuta Imamura et al., "Web Access Monitoring Mechanism via Android Web View for Threat Analysis," *International Journal of Information Security*, vol. 20, no. 6, pp. 833-847, 2021. [CrossRef] [Google Scholar] [Publisher Link]
[19] Andrei-Claudiu Veres, and Andrei Dumitriu, "The Privacy and Security Risks of Mobile In-App Browsers," *SC@RUG 2023 proceedings*, pp. 19-23, 2023. [Google Scholar]
[20] Mohamed A. El-Zawawy, Eleonora Losiouk, and Mauro Conti, "Vulnerabilities in Android Webview Objects: Still Not the End!" *Computers and Security*, vol. 109, 2021. [CrossRef] [Google Scholar] [Publisher Link]
[21] Ksenia Peguero, and Xiuzhen Cheng, "Electrolint and Security of Electron Applications," *High-Confidence Computing*, vol. 1, no. 2, pp. 1-14, 2021. [CrossRef] [Google Scholar] [Publisher Link]
[22] Com.Basecamp.bc3 Webview Javascript Injection and JS Bridge Takeover, Hackerone, 2021. [Online]. Available: https://hackerone.com/reports/1343300
[23] Zomato for Business Android, Vulnerability in Exported Activity WebView, Hackerone, 2019. [Online]. Available: https://hackerone.com/reports/537670
[24] Vulnerability in TikTok Android App Could Lead to One-Click Account Hijacking, Microsoft, 2022. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2022/08/31/vulnerability-in-tiktok-android-app-could-lead-to-one-click-account-hijacking/
[25] Security Checklist, Android Developers. [Online]. Available: https://developer.android.com/privacy-and-security/security-tips
[26] iOS Platform APIs, OWASP Mobile Application Security. [Online]. Available: https://mas.owasp.org/MASTG/0x06h-Testing-Platform-Interaction/

**Glossary:**

The glossary for this article defines technical terms related to WebView security. Here are some of the important terms:

- **WebView**: A component used in mobile apps to display web content.
- **Cross-Site Scripting (XSS):** A vulnerability that allows attackers to inject malicious code into web pages.
- **Remote Code Execution (RCE):** A vulnerability that allows attackers to run arbitrary code on a victim's device.
- **Content Security Policy (CSP):** A security measure that restricts the sources from which web content can be loaded.
- **Secure Sockets Layer (SSL)/Transport Layer Security (TLS):** A cryptographic protocol that ensures secure communication between a web server and a browser.

# Appendix 1:

### Android Secure Code sample:

```
public class SecureWebViewActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_secure_webview);

        WebView webView = findViewById(R.id.webview);
        WebSettings webSettings = webView.getSettings();

        // Basic security settings
        webSettings.setJavaScriptEnabled(false); // Enable only if necessary
        webSettings.setAllowFileAccess(false);
        webSettings.setAllowFileAccessFromFileURLs(false);
        webSettings.setAllowUniversalAccessFromFileURLs(false);

        // Use Safe Browsing
        WebView.startSafeBrowsing(this, success -> {
            if (!success) {
                Log.e("WebView", "Safe Browsing initialization failed!");
            }
        });

        // Restrict URL loading
        webView.setWebViewClient(new WebViewClient() {
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, String url) {
                return !url.startsWith("https://trusted.com"); // Allow only trusted URLs
            }

            @Override
            public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
                handler.cancel(); // Reject unsafe SSL certificates
            }
        });

        // Load a trusted URL
        webView.loadUrl("https://trusted.com");
    }
}
```

# Appendix 2:

### iOS Secure Code example:

```
import UIKit
import WebKit

class SecureWebViewController: UIViewController, WKNavigationDelegate, WKUIDelegate {

    var webView: WKWebView!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Configure WKWebView
        let webViewConfiguration = WKWebViewConfiguration()
        webViewConfiguration.preferences.javaScriptEnabled = false // Disable JavaScript by default
        webViewConfiguration.preferences.javaScriptCanOpenWindowsAutomatically = false // Prevent pop-ups

        // Create WKWebView
        webView = WKWebView(frame: self.view.bounds, configuration: webViewConfiguration)
        webView.navigationDelegate = self
        webView.uiDelegate = self

        self.view.addSubview(webView)
```

```
        // Load content securely
        if let url = URL(string: "https://trusted-domain.com") {
            let request = URLRequest(url: url)
            webView.load(request)
        }
    }

    // WKNavigationDelegate - Restrict navigation to trusted domains
    func webView(_ webView: WKWebView, decidePolicyFor navigationAction: WKNavigationAction, decisionHandler: @escaping
(WKNavigationActionPolicy) -> Void) {
        if let url = navigationAction.request.url {
            let trustedDomains = ["trusted-domain.com"]
            if trustedDomains.contains(url.host ?? "") {
                decisionHandler(.allow) // Allow navigation for trusted domains
            } else {
                print("Blocked navigation to: \(url)")
                decisionHandler(.cancel) // Block navigation for untrusted domains
            }
        } else {
            decisionHandler(.cancel)
        }
    }

    // WKNavigationDelegate - Handle SSL errors (additional checks beyond built-in)
    func webView(_ webView: WKWebView, didFailProvisionalNavigation navigation: WKNavigation!, withError error: Error) {
        print("Navigation failed with error: \(error.localizedDescription)")
    }

    // Clear sensitive data from cache
    override func viewWillDisappear(_ animated: Bool) {
        super.viewWillDisappear(animated)
        webView.configuration.websiteDataStore.httpCookieStore.getAllCookies { cookies in
            cookies.forEach { cookie in
                self.webView.configuration.websiteDataStore.httpCookieStore.delete(cookie)
            }
        }
        webView.configuration.websiteDataStore.removeData(ofTypes: WKWebsiteDataStore.allWebsiteDataTypes(), modifiedSince: Date.distantPast) {}
    }

    // Example of Secure Communication (WKScriptMessageHandler)
    func setupMessageHandler() {
        webView.configuration.userContentController.add(self, name: "secureHandler")
    }
}

// Extension to handle JavaScript messages securely
extension SecureWebViewController: WKScriptMessageHandler {
    func userContentController(_ userContentController: WKUserContentController, didReceive message: WKScriptMessage) {
        guard message.name == "secureHandler" else { return }
        print("Received message from web content: \(message.body)")
    }
}
```