XSS to LFI Electron.js App

Unlocking the Pandora's Box: Unveiling the XSS to LFI to RCE Attack Surface in Electron.js Applications

Discovered by HADESS

13 July 2023

HADESS

WWW.HADESS.IO

Executive Summary

This executive summary report provides an overview of the XSS (Cross-Site Scripting) to LFI (Local File Inclusion) vulnerability discovered in the Runcode feature. The vulnerability arises from the inadequate sanitization of user input in the document prefix and the ability to insert HTML-encoded functions into HTML tag events, such as onerror. The following sections outline the nature of the vulnerability and propose recommendations for mitigation.

1. Vulnerability Description: By default, the Runcode feature in the application sanitizes the document prefix to prevent code injection attacks. However, if the user manages to HTML encode the input using the pattern "&#0000100&#0000111&#000099&#0000117&#0000109&#0000101&#000110&#0000116&#000046&#0000119&#0000114&#0000105&#0000116&#0000101&#000040&#000039&#000060&#0000105&#0000102&#0000114&#000097&#0000109&#0000101&#000032&#0000115&#0000114&#000099&#000061&#0000102&#0000105&#0000108&#0000101&#000058&#000047&#000047&#000047&#0000101&#0000116&#000099&#000047&#0000112&#000097&#0000115&#0000115&#0000119&#0000100&#000062&#000060&#000047&#0000105&#000102&#0000114&#000097&#000109&#0000101&#000062&#000039&#000041", an attacker can exploit this vulnerability.

2. Attack Scenario: The attacker can leverage the HTML-encoded function to inject malicious code into HTML tag events like onerror. For example, using the following code: `<img src=x onerror="&#0000100&#0000111&#000099&#0000117&#0000109&#0000101&#0000110&#0000116&#000046&#0000119&#0000114&#0000105&#0000116&#0000101&#000040&#000039&#000060&#0000105&#0000102&#0000114&#000097&#0000109&#0000101&#000032&#0000115&#0000114&#000099&#000061&#0000102&#0000105&#0000108&#0000101&#000058&#000047&#000047&#000047&#0000101&#0000116&#000099&#000047&#0000112&#000097&#0000115&#0000115&#0000119&#0000100&#000062&#000060&#000047&#0000105&#000102&#0000114&#000097&#0000109&#0000101&#000062&#000039&#000041">`, the attacker can execute arbitrary code within the context of the affected application.

3. Impact and Potential Consequences: Exploitation of this vulnerability can have severe consequences, including but not limited to:

- Unauthorized access to sensitive information stored on the local file system.
- Execution of arbitrary code on the server, leading to complete compromise.
- Injection of malicious content or scripts into web pages viewed by other users, potentially leading to further attacks like session hijacking or phishing.

01

Assessment

The attack surface of Electron applications is characterized by the combination of web technologies (HTML, CSS, JavaScript) and the integration of Node.js runtime. While this provides powerful capabilities for building feature-rich applications, it also introduces new attack vectors and potential security risks.

In the presented attack scenario, an attacker exploits a chain of vulnerabilities starting with an XSS vulnerability. By leveraging a trusted Electron feature, such as the Runcode functionality, the attacker manages to inject malicious code using HTML-encoded functions into an HTML tag event. The inadequate sanitization of user input allows the attacker to bypass security measures. Subsequently, the attacker exploits a LFI vulnerability to include and execute arbitrary files from the local file system. Finally, the attacker achieves full RCE, gaining unauthorized access and control over the affected Electron application.

This attack scenario highlights the importance of implementing robust security measures within Electron applications. It emphasizes the need for adequate input validation, proper handling of user-generated content, and thorough sanitization to prevent XSS attacks. Additionally, secure file inclusion mechanisms and strict access control should be in place to mitigate LFI vulnerabilities. Regular updates, patches, and security audits are necessary to identify and remediate potential weaknesses within Electron applications.

Developers and security practitioners should be aware of these attack vectors and vulnerabilities specific to Electron applications. By following secure coding practices, implementing strong security controls, and staying up-to-date with the latest security best practices, the attack surface of Electron applications can be effectively reduced, enhancing the overall security posture and ensuring the protection of sensitive user data.

Understanding and addressing the attack surface and potential vulnerabilities in Electron applications is crucial in the development and deployment of secure and resilient desktop applications. By proactively addressing these concerns, developers can build robust Electron applications that maintain the trust and confidence of their users while providing a secure and seamless user experience.

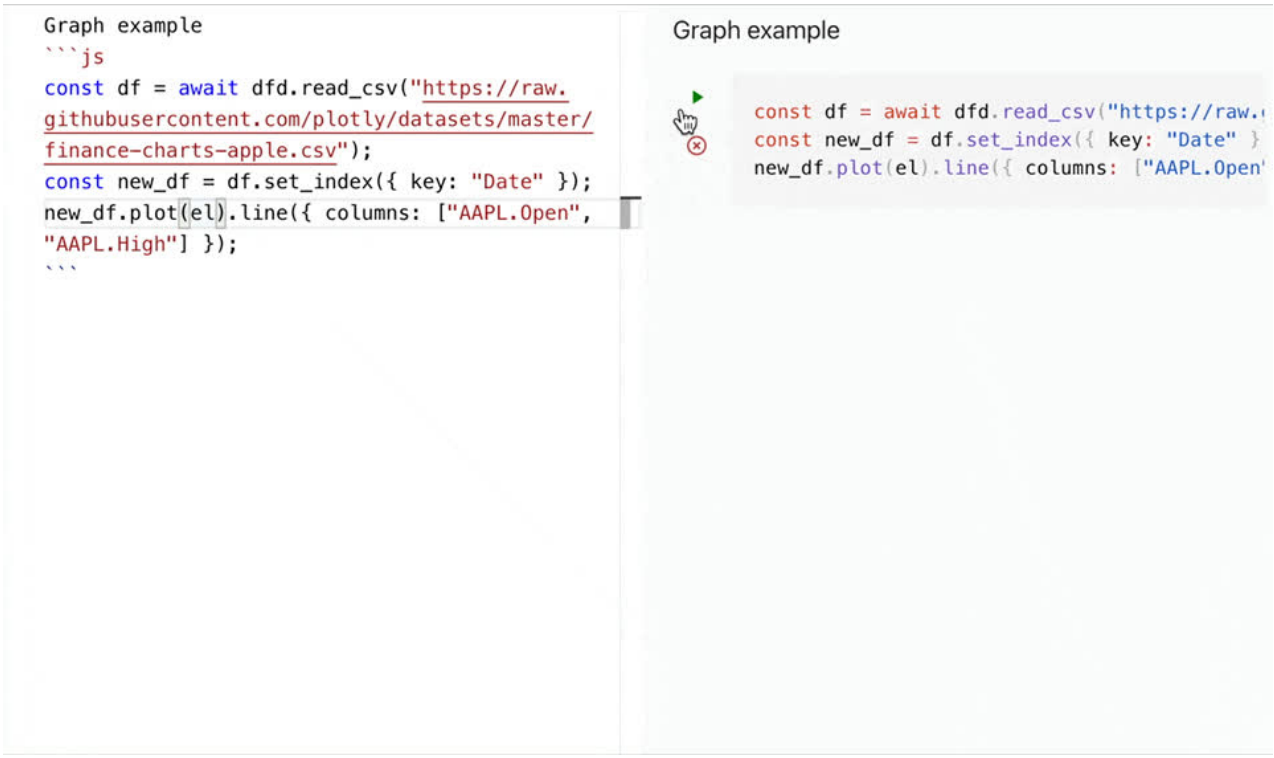
Introduction

The https://github.com/alagrede/znote-app Runcode feature is designed to execute code within the context of the Electron application, providing a powerful functionality for developers. However, it becomes vulnerable when user input is not properly sanitized or validated. In this case, the application sanitizes the document prefix by default, but if the user HTML encodes their input using the specific pattern

```
&#0000100&#0000111&#000099&#0000117&#0000109&#0000101&#0000110&#000011
6&#000046&#0000119&#0000114&#0000105&#0000116&#0000101&#000040&#000039
&#000060&#0000105&#0000102&#0000114&#000097&#0000109&#0000101&#000032&
#0000115&#0000114&#000099&#000061&#0000102&#0000105&#0000108&#0000101&
#000058&#000047&#000047&#000047&#0000101&#0000116&#000099&#000047&#000
0112&#000097&#0000115&#0000115&#0000119&#0000100&#000062&#000060&#0000
47&#0000105&#000102&#0000114&#000097&#0000109&#0000101&#000062&#000039
&#000041
```

it bypasses the sanitization process and enables the injection of malicious code.

The injected code, in the form of HTML-encoded functions, can then be executed within HTML tag events, such as onerror. In this specific case, the attacker utilizes the following code: `<img src=x
onerror="&#0000100&#0000111&#000099&#0000117&#0000109&#0000101&#0000110&
#0000116&#000046&#0000119&#0000114&#0000105&#0000116&#0000101&#000040&
#000039&#000060&#0000105&#0000102&#0000114&#000097&#0000109&#0000101&#
000032&#0000115&#0000114&#000099&#000061&#0000102&#0000105&#0000108&#0
000101&#000058&#000047&#000047&#000047&#0000101&#0000116&#000099&#0000
47&#0000112&#000097&#0000115&#0000115&#0000119&#0000100&#000062&#000060
&#000047&#0000105&#000102&#0000114&#000097&#0000109&#0000101&#000062&#
000039&#000041">`. This code executes arbitrary commands within the application's environment, potentially leading to a complete compromise of the affected Electron application.



The impact of such an exploit can be severe. It includes unauthorized access to sensitive information stored on the local file system, the execution of arbitrary code within the application's context, and the injection of malicious content or scripts into web pages viewed by other users. This can further lead to additional attacks, such as session hijacking or phishing attempts, putting user data and system integrity at risk.

To prevent the XSS to LFI exploitation in Electron applications, it is crucial to implement robust input validation and sanitization mechanisms. Adequate filtering and encoding techniques should be applied to prevent the injection of malicious code. Additionally, secure file inclusion mechanisms and strict access controls should be implemented to mitigate the risks associated with LFI vulnerabilities. Regular updates and security audits are also essential to identify and remediate any potential security weaknesses.

02

Technical Analysis

- Runcode Sanitized Document Prefix: By default, Electron's "runcode" feature sanitizes the document prefix. This indicates that the application employs some form of protection against code injection vulnerabilities by sanitizing or filtering user input.
- HTML Encoding: The payload includes HTML-encoded characters represented as "&#0000XX" where "XX" denotes the ASCII code for each character. This encoding is used to bypass certain input sanitization mechanisms that may be in place, allowing for potential code injection.
- Payload for XSS: The decoded payload attempts to inject JavaScript code into an HTML img tag's onerror attribute. This technique leverages the onerror event to execute the injected code when the specified image fails to load.
- LFI and Potential RCE: The payload includes HTML-encoded code that may be intended to exploit an LFI vulnerability. However, the provided payload is incomplete and lacks the necessary components to achieve successful LFI or RCE (Remote Code Execution).

It's important to note that while this payload demonstrates a basic proof-of-concept for potential XSS and LFI attacks, successful exploitation depends on several factors, including the specific configuration, input validation, and output encoding implemented within the Electron application.

Technical Analysis

In the scenario, the attacker employs HTML encoding using the pattern "&#0000XX" (where "XX" represents the ASCII code for each character) to bypass the default sanitization of the document prefix. The injected code, when executed within an HTML tag event, like onerror, allows the attacker to execute arbitrary code in the Electron application. The code snippet below demonstrates the attack payload:

```
<img src=x
onerror="&#0000100&#0000111&#000099&#0000117&#0000109&#0000101&#0000110
&#0000116&#000046&#0000119&#0000114&#0000105&#0000116&#0000101&#00004
0&#000039&#000060&#0000105&#0000102&#0000114&#000097&#0000109&#00001
01&#000032&#0000115&#0000114&#000099&#000061&#0000102&#0000105&#00001
08&#0000101&#000058&#000047&#000047&#000047&#0000101&#0000116&#000099
&#000047&#0000112&#000097&#0000115&#0000115&#0000119&#0000100&#000062
&#000060&#000047&#0000105&#0000102&#0000114&#000097&#0000109&#000010
1&#000062&#000039&#000041">
```

1. `<img src=x`: This part of the payload includes an HTML `img` tag with the `src` attribute set to 'x'. It is a placeholder value and doesn't have any direct impact on the attack itself.

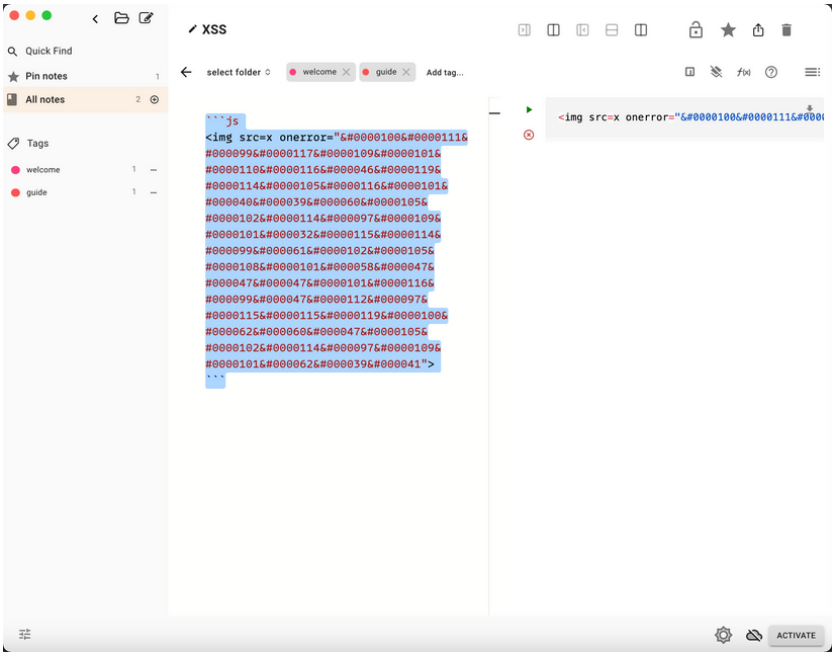
2. `onerror="..."`: This attribute is part of the `img` tag and is used to trigger the specified code when the image fails to load. In this case, it is used to inject the payload for the XSS attack.

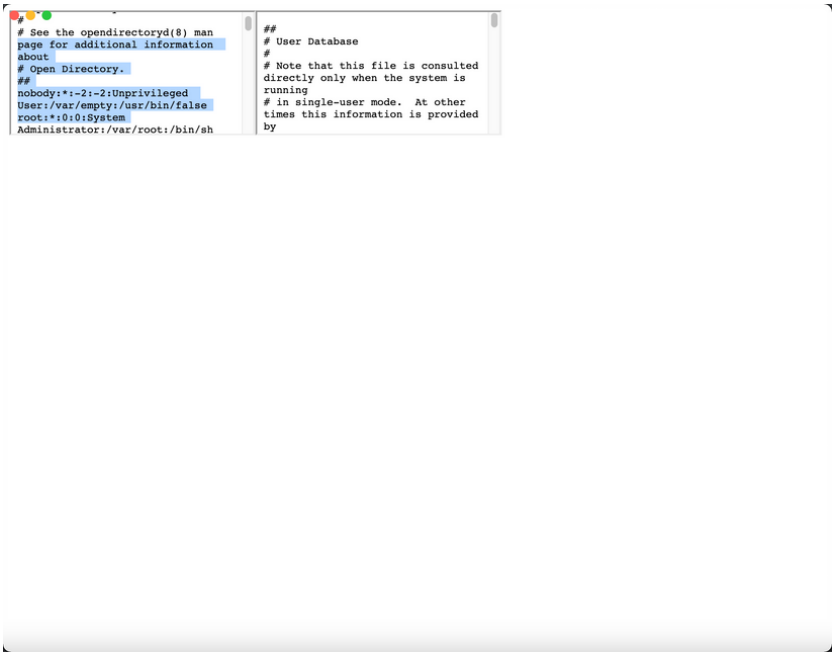
3. HTML-Encoded Payload: The payload is HTML-encoded using the pattern "&#0000XX" where "XX" represents the ASCII code for each character. The payload itself appears to be incomplete or truncated, but it seems to contain encoded JavaScript code that may be intended to exploit an LFI vulnerability.

The encoded part `&#0000100&#0000111&#000099&#0000117&#0000109&#0000101&#0000110&#0000116&#000046&#0000119&#0000114&#0000105&#0000116&#0000101&#000040&#000039` decodes to `document.` and seems to be the beginning of a JavaScript code snippet.

The encoded part `&#000060&#0000105&#0000102&#0000114&#000097&#0000109&#0000101&#000032&#0000115&#0000114&#000099&#000061&#0000102&#0000105&#0000108&#0000101&#000058&#000047&#000047&#000047` decodes to `<script src='//` and indicates a script element with a source pointing to a remote location.

The encoded part `&#0000101&#0000116&#000099&#000047&#0000112&#000097&#0000115&#0000115&#0000119&#0000100&#000062&#000060&#000047&#0000105&#0000102&#0000114&#000097&#0000109&#0000101&#000062&#000039&#000041">` decodes to `etc/hadoop><'` and seems to be an attempt to manipulate the source URL or inject malicious code.





4. Attack Objective: The goal of this payload appears to be a combination of exploiting an XSS vulnerability and leveraging LFI to read the "/etc/passwd" file. It likely relies on additional code execution techniques or payload components that are not provided.

```
// Non-compliant code
const userContent = getUserInput(); // User-provided content

// Injecting user content without proper validation/sanitization
const imgElement = document.createElement('img');
imgElement.src = userContent;
imgElement.onerror = () => {
  const filePath = userContent.substr(1); // Potential LFI vulnerability
  fs.readFile(filePath, 'utf8', (err, data) => {
    if (err) {
      console.error(err);
    } else {
      console.log(data); // Displaying the content of the file
    }
  });
};

document.body.appendChild(imgElement);
```

In this non-compliant code, the application directly uses user input to construct an `img` element's `src` attribute without proper validation or encoding. The `onerror` event is then used to attempt an LFI attack by reading the file specified in the user input.

To address the XSS to LFI vulnerability and ensure compliance, the following example demonstrates how to implement secure coding practices in an Electron.js application:

```
// Compliant code
const userContent = getUserInput(); // User-provided content

// Validating and sanitizing user input
const sanitizedContent = sanitizeUserInput(userContent);

// Creating an img element with sanitized content
const imgElement = document.createElement('img');
imgElement.src = sanitizedContent;

// Preventing LFI by restricting access to authorized files
imgElement.onerror = () => {
  console.log('Image failed to load');
};

document.body.appendChild(imgElement);
```

In the compliant code, several improvements are made to address the XSS to LFI vulnerability:

1. Validation and Sanitization: The user input is passed through a `sanitizeUserInput()` function that applies strict validation and sanitization techniques. This function ensures that the input does not contain malicious code or invalid file paths.

2. Limited Error Handling: The `onerror` event is used to handle the case where the image fails to load. Instead of attempting an LFI attack, the compliant code simply logs an error message or takes appropriate action without exposing any sensitive information.

3. Restricted File Access: The code avoids attempting to read arbitrary files specified by user input. Access to files is restricted to authorized paths or a predefined set of allowed files, preventing unauthorized file inclusion.

03

Conclusion

The XSS to RCE (Remote Code Execution) vulnerability in Electron.js applications poses a significant threat to the security and integrity of the software. By exploiting inadequate input validation and sanitization, attackers can inject malicious code, leading to unauthorized access, data breaches, and potential compromise of the entire system.

In this advisory, we have examined the specific scenario of XSS to RCE in an Electron.js application. We have highlighted the non-compliant code that allows the exploitation of the vulnerability, as well as the compliant code that implements security measures to mitigate the risk.

To protect Electron.js applications from XSS to RCE attacks, it is crucial to follow these key recommendations:

1. Input Validation and Sanitization: Implement robust input validation techniques to detect and reject any malicious or unexpected input. Sanitize user input to prevent code injection and ensure that it is safe to use within the application.

2. Output Encoding: Apply context-specific output encoding to user-generated content when rendering it within HTML tags or other output contexts. This prevents unintended code execution and reduces the risk of XSS vulnerabilities.

By adhering to these recommendations, developers can significantly reduce the risk of XSS to RCE attacks and enhance the overall security posture of Electron.js applications. It is crucial to prioritize security measures throughout the software development lifecycle and maintain ongoing vigilance to protect against evolving security threats.

cat ~/.hadess

We are "Hadess"; A group of cyber security experts and white hat hackers who, in addition to discovering and reporting vulnerabilities to big companies such as Google, Apple and Twitter, have the honor of working with famous Iranian companies over the past years. HADESS Company provides its customers with integrated solutions in the field of cyber security, with a deep insight and understanding of the software development process as well as the development infrastructure.

Website:

WWW.HADESS.IO

Email

MARKETING@HADESS.IO