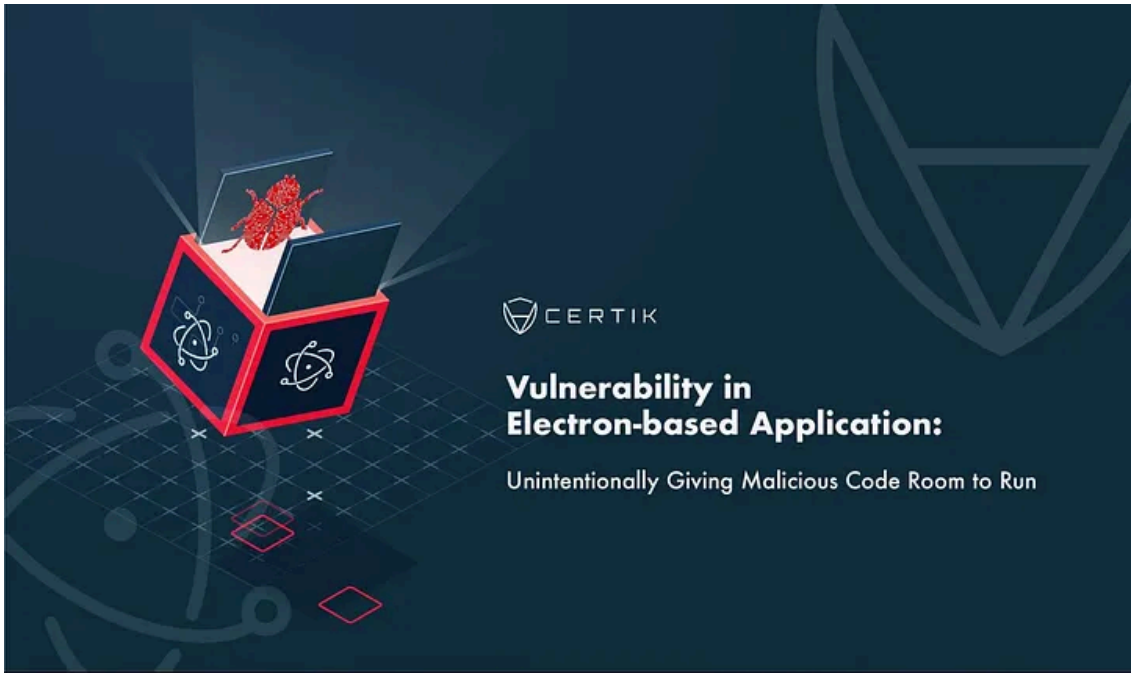
Vulnerability in Electron-based Application: Unintentionally Giving Malicious Code Room to Run

By CertiK

Published: 2020-07-02 · Archived: 2026-04-05 16:57:26 UTC

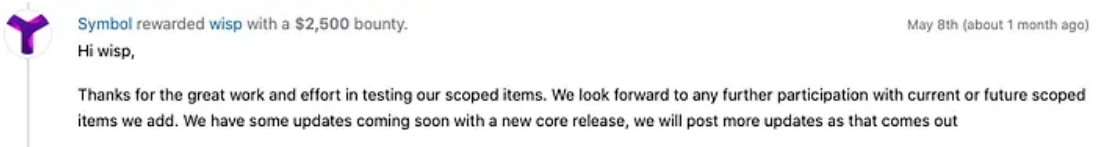
Press enter or click to view image in full size



Background

One of our security engineers discovered a remote code execution vulnerability in the Symbol desktop wallet and reported the vulnerability through their bug bounty program. Given the nature of the issue, the Symbol team took immediate action to update their code, and a fix was deployed in the v0.9.11 release.

Press enter or click to view image in full size



Though the HackerOne report is not yet public, we give many thanks to the Symbol team for allowing us to disclose and share our findings.

The Symbol wallet is an Electron-based desktop application, and the vulnerability we found was in the Electron configuration itself. Before jumping into the finding, let's briefly review what Electron is and the security aspects of its application.

What is Electron?

Electron is an open-source software framework developed and maintained by GitHub, enabling developers to build cross-platform desktop applications with web technologies such as HTML, CSS, and Javascript. Electron combines the Chromium rendering engine and Node.js into a single runtime. Some well-known Electron applications include Atom editor, Visual Studio Code, and Slack.

There are a couple benefits to using Electron:

- Web developers can build cross-platform desktop applications that run on different operating systems with major Javascript framework libraries including Angular, React and Vue-all without needing to spend the extra time to learn new programming languages.
- Debugging an Electron-based application is easier than traditional desktop applications. The DevTools Extension in Chromium allows developers to debug their Electron-based application the same way as a web application.

Electron security and the danger of Node.js

Electron-based applications are essentially web applications, so they contain common web vulnerabilities such as cross-site scripting (XSS), SQL injection, and authentication and authorization flaws.

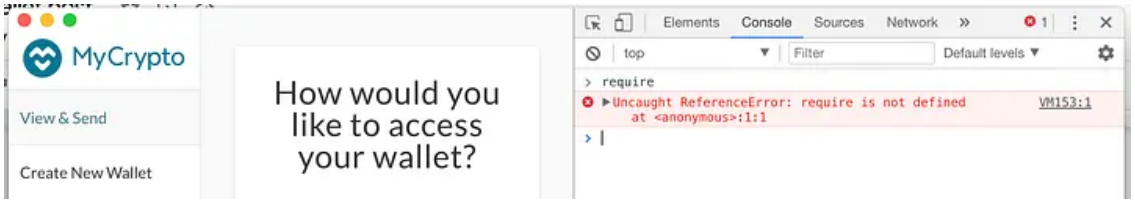
Electron comes with a number of APIs to support the development of desktop applications, and on top of that, it can also use Node.js modules. The access to Node.js modules allows Electron-based desktop applications to support more features than regular web applications that run in a web browser. However, enabling Node.js comes with greater security risks. For example, attackers can execute system commands on the victim's machine if they can find a way to inject malicious Javascript into the application.

Checking if Node.js is enabled

To check whether Node.js is enabled in the Electron application, users can send the module import function `require` in the development console. The development console can be opened in Chrome with "option+command+i" on macOS.

If Node.js is disabled, the console will return an error message, "require is not defined", as seen in the following screenshot:

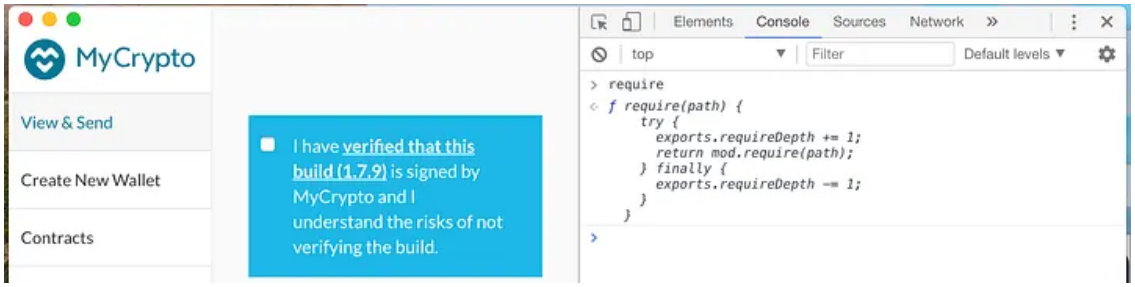
Press enter or click to view image in full size



When Node.js is disable

But if Node.js is enabled, users will see the following:

Press enter or click to view image in full size



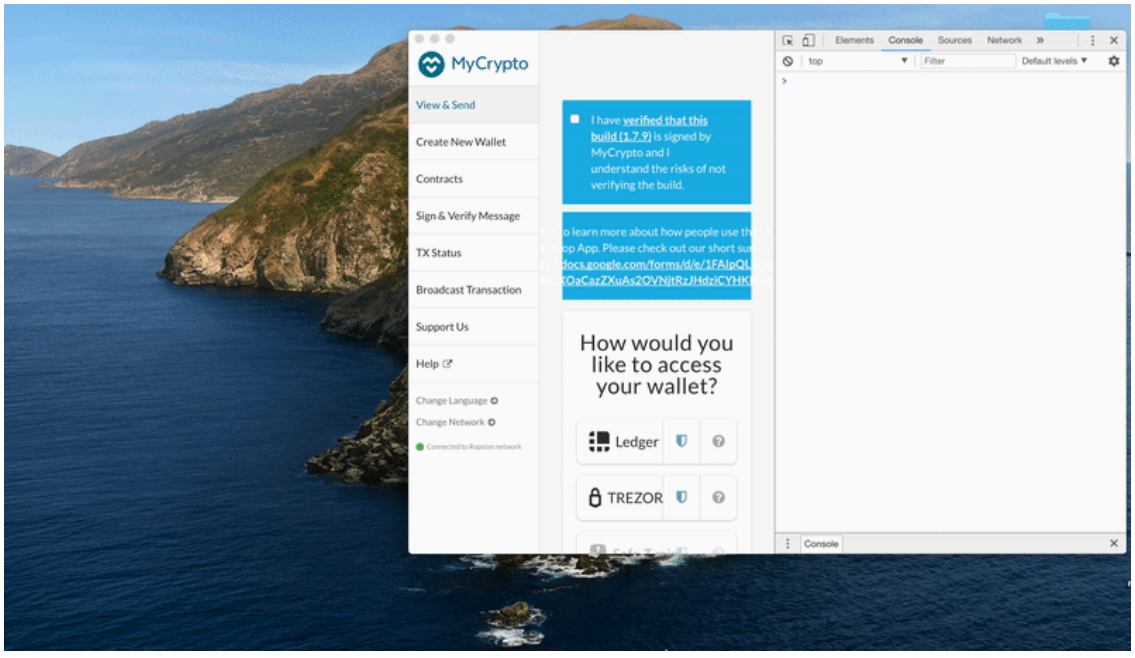
When Node.js is enabled

The danger with enabling Node.js is that it leaves an opening for hackers to execute malicious system commands by injecting Javascript code.

For an example of how this would work, you can try sending the following command in the development console to open the calculator program in macOS:

```
require('child_process').exec('/System/Applications/Calculator.app/Contents/MacOS/Calculator')
```

Press enter or click to view image in full size



To mitigate the potential risk of system code execution caused by injected Javascript, starting from version 5.0.0, Electron has disabled access to the Node.js function by default. Developers can re-enable access to the Node.js function by setting the `nodeIntegration` to true in the build configuration file (though this is not recommended!).

Note that in 2018, a critical vulnerability was also discovered in Electron that allows attackers to re-enable Node.js integration at runtime, which would potentially lead to remote code execution. This blog explains more in detail. The point here is that it's important to keep your application up-to-date with the latest Electron framework release.

How to exploit the Symbol wallet remote code execution vulnerability

Now that we know one of the items to look for when testing an Electron-based application, let's dive into the vulnerability we discovered in the Symbol wallet.

The Symbol desktop wallet is open-source, and the source code for the application can be found in their Github repository.

Get CertiK's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

The build.js file is the Electron build configuration file for their application. The code snippet below checks if the application runs on *Darwin* (macOS); if not, app.on creates a new browser window with the `createWindow` function.

```
....code...
  if (process.platform === 'darwin') {
      app.on('ready', createMac)
    } else {
       app.on('ready', createWindow)....code...
```

In the `createwindow` function, the `windowOptions` variable inside the function contains browser window configuration options. Notice the line highlighted in red-the `nodeIntegration` variable is set to true, which enables Node.js in the Electron application:

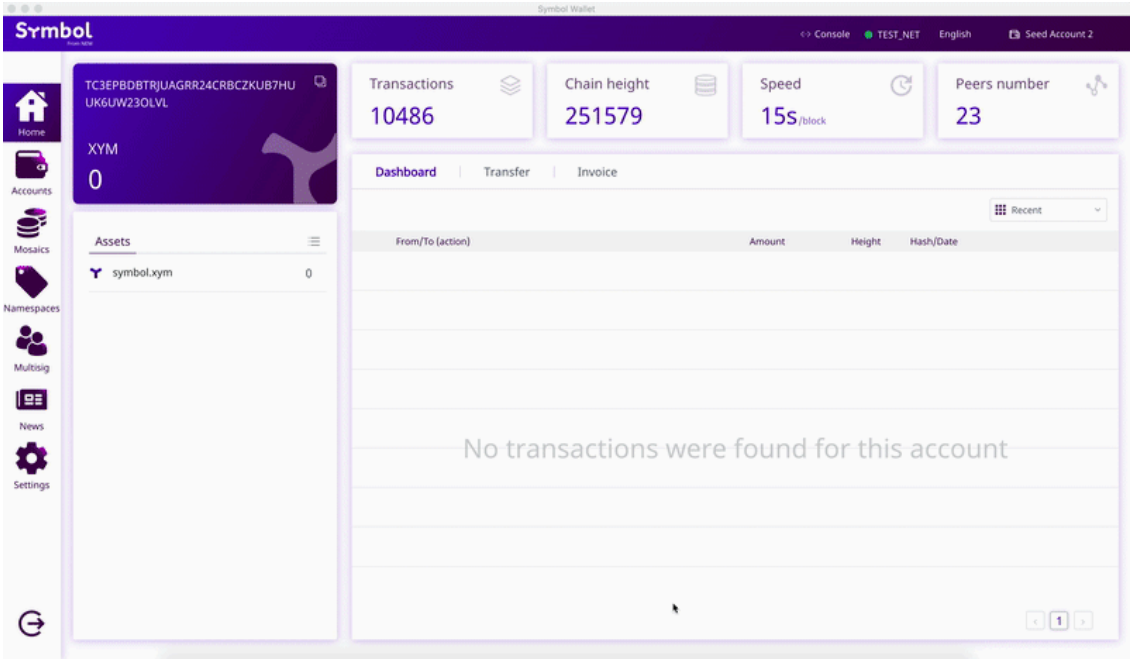
```
...code...
function createWindow() {
    const windowOptions = {
      minWidth: width,
      minHeight: height,
      width: width,
      height: height,
```

```
      title: app.getName(),
      titleBarStyle: 'hiddenInset',
      webPreferences: {
        nodeIntegration: true,
      },
      resizable: true,
    }
....code...
    mainWindow = new BrowserWindow(windowOptions)
}
```

Based on the build.js configuration file, we learn that if the application runs on the Windows operating system, Node.js will be enabled. In order to take advantage of the enabled Node.js, an attacker would need to find a way to execute arbitrary Javascript in the application. This can often be achieved by exploiting the XSS (Cross Site Scripting) vulnerability, or a user unknowingly loads a remote web page embedded with Javascript that a hacker controls from within the Electron application.

Luckily for us, the Symbol desktop wallet (release v9.7) provides a feature to view “News”. Once the user clicks a link in the news feed, the application will navigate away from the wallet interface and loads the external website (Github, in this case) inside the current window.

Press enter or click to view image in full size



Show me the exploit!

To demonstrate, you can host the code snippet below on your website; it will be a trivial task to inject the URL to a website over Github. With `nodeIntegration` set to true and Node.js enabled, arbitrary javascript execution can be escalated to remote code execution with the help of the "child_process" module.

After the user visits the infected page and clicks the “Close” button, the calculator will open on the user’s computer. The calculator itself is harmless, but in this example, the fact that it even opened in the target system means that the application was vulnerable and successfully exploited.

To see the exploit in action, check out the proof-of-concept video: https://www.youtube.com/watch?v=X5R2xC3Jcy0

Proof-of-Concept code snippet:

```
<!DOCTYPE html><h1>click me</h1>
<button type="button" onClick="rce_calc()">Submit</button>
<script>
  function rce_calc(){
  const { exec } = require('child_process');
  exec('calc');
}
</script>
```

This example presents the steps of the exploit pretty obviously; the payload requires the user to click the button to trigger the system command. In reality, an attacker may host a malicious script that triggers the system command execution automatically and inconspicuously when users visit the page.

Symbol resolved the issue that we detected by setting `nodeIntegration` to false, which disabled javascript to access Node.js function. This change is reflected in their current build.js file. They have also updated the "News" feature to prohibit the loading of remote websites into the Electron window.

Exploring other Electron-based cryptocurrency wallets

As a security researcher, when you exploit a vulnerability in one application, you always want to see if the same type of vulnerability exists elsewhere. With just a quick search, we found another notable Electron-based cryptocurrency wallet: MyCrypto.

At the time of testing, we discovered MyCrypto had `nodeIntegration` set to true and Node.js enabled. Though we didn't find a way to exploit the vulnerability with cross-site scripting or arbitrary page redirection inside the application, we know best practice is to prevent hackers from turning "self-xss" into a command code execution.

True to our and CertiK’s core values, we reported the issue to MyCrypto as well via their Github repository: https://github.com/MyCryptoHQ/MyCrypto/issues/3261

After we reported the issue, MyCrypto stated the vulnerability will be fixed in the next release.

Takeaways and Lessons Learned

With the time we spent learning the ins-and-outs of the Electron framework, we’ve put together a quick list that you can reference to improve the security of Electron-based applications:

- Remove access to the development console in any production releases.
- Set `nodeIntegration` to false unless the application absolutely requires it.
- Disallow the application from navigating away from the main application with `event.preventDefault()`.
- Develop the application in React, Vue, or Angular (2+) to minimize the chance of a XSS (cross site scripting) vulnerability.
- Keep your application up-to-date with the latest Electron framework release.
- Regularly review the official security guidance when developing your Electron application, which contains helpful security recommendations.

Additionally, performing security audits and penetration tests, whether by an internal security team or third-party firm, are important to ensure the security of your system. Security professionals will attempt to break the system with a malicious hacker's mindset, helping identify and remediate vulnerabilities before a bad actor exploits them.

CertiK is deep-rooted in academia, but our strength is in taking the research that's available and applying them to real-world situations. We aim to provide value with our security-first philosophy through the use of rigorous techniques, creative thinking, and adaptability. It is our goal and responsibility to contribute to the crypto & blockchain community and help companies secure their users' assets for a more secure experience.

If you're interested in getting an external team to check the security of your systems, contact us for a proposal and quote at **bd@certik.io**

References

https://www.electronjs.org/docs/tutorial/security
