WHITEBOX CODE AUDIT REPORT

for

COFIX COMMUNITY

Prepared By: Shuxiao Wang

Nov. 26, 2020

Document Properties

Client	CoFiX Community
Report	Whitebox Code Audit Report
Target	CoFiX
Version	1.0
Author	Darker
Auditors	Darker
Reviewer	Chiachih Wu
Approver	Xuxian Jiang
Classification	Confidential

Version info

Version	Date	Auditors	Description
1.0	Nov. 26, 2020	Darker	Final Release
0.1	Nov. 17, 2020	Darker	Initial Draft

Contact

For more detailed information regarding this report, please contact with PeckShield [2] 。

Contact	Shuxiao Wang
Phone number	+86 173 6454 5338
E-mail	contact@peckshield.com

Contents

1 Introduction 4
- 1.1 About CoFiX . . . 4
- 1.2 Code Audit Scope . . . 4

2 Code Audit Procedures and Explanation 5
- 2.1 Vulnerable Third-Party Dependencies . . . 5
- 2.2 Third-Party JavaScript . . . 5
- 2.3 XSS Vulnerability . . . 5
- 2.4 Inadequate HTTP Security Headers . . . 8
- 2.5 Inadequate Cross-Origin Resource Sharing (CORS) Settings . . . 8
- 2.6 Wallet Connection Security . . . 9
- 2.7 Insecure Session Management . . . 10
- 2.8 Cache Manipulation and Poisoning . . . 10
- 2.9 Authentication and Authorization Bypass . . . 10

3 Conclusion 11

Reference 12

1 Introduction

The purpose of this code audit operation is to perform a comprehensive security assessment against CoFiX Protocol web user interface functionality and address any potential cyber security risks.

1.1 About CoFiX

CoFiX Protocol, our code audit target, is an advanced automated market-making protocol. The provided codebase used in this audit covers the web user interface functionality.

- Source code of CoFiX Protocol Web UI (written with Typescript)
- Web service URL: https://cofix.io/

1.2 Code Audit Scope

- Cross-Site Scripting.
- Cross-Site Request Forgery.
- Inadequate HTTP response headers.
- Authentication and authorization bypass (if applicable).
- Application business logic validation bypass.
- Cache manipulation and poisoning.
- Inadequate Cross-Origin Resource Sharing (CORS) settings.
- Insecure Session Management.
- Vulnerable third-party dependencies.

2 Code Audit Procedures and Explanation

2.1 Vulnerable Third-Party Dependencies

Vulnerable third party dependencies are often ignored during security audit. However, certain type of vulnerable third party dependencies/libraries can cause severe damage to software system. To verify if dependencies in target system have any known vulnerability, we cross checked the listed dependencies in `package.json` along with their version number, as shown in Figure 2.1. No known vulnerability was identified.

2.2 Third-Party JavaScript

Except Google Analytics JavaScript, we notice that two other third-party JavaScript files are loaded as illustrated in Figure 2.2. Since loading remote JS files could be an attack surface which leads to financial loss, we highly recommend to store all needed JS files onto Amazon S3 buckets and load them from local storage.

2.3 XSS Vulnerability

To detect if any XSS vulnerabilities exist in target system, we checked through all HTML template files, as shown in Figure 2.3, in provided source code and went through variable input locations, which is where the XSS normally happens.

To explain how XSS detection procedure works, we use `banner.page.html` template file as an example. Figure 2.4 shows that the `banner.page.html` file has three input variables which may potentially abused by malicious users, leading to XSS vulnerability. After the browser executes and renders the downloaded JS files, the final content is rendered as Figure 2.5.

To verify if such input variables can eventually lead to XSS vulnerabilities, we further track the rendered variables and check if the variable values can be controlled by user. In our `banner.page.html`

```
"dependencies": {
  "@angular/common": "~10.0.0",
  "@angular/core": "~10.0.0",
  "@angular/forms": "~10.0.0",
  "@angular/platform-browser": "~10.0.0",
  "@angular/platform-browser-dynamic": "~10.0.0",
  "@angular/router": "~10.0.0",
  "@capacitor/core": "2.4.2",
  "@datorama/akita": "^4.22.0",
  "@ionic-native/core": "^5.0.0",
  "@ionic-native/splash-screen": "^5.0.0",
  "@ionic-native/status-bar": "^5.0.0",
  "@ionic/angular": "^5.0.0",
  "@ngx-translate/core": "^13.0.0",
  "@ngx-translate/http-loader": "^6.0.0",
  "bignumber.js": "^9.0.1",
  "ethers": "^5.0.15",
  "ngx-cacheable": "^1.4.3",
  "rxjs": "~6.5.5",
  "tslib": "^2.0.0",
  "zone.js": "~0.10.3"
},
```

Figure 2.1: CoFix Dependencies

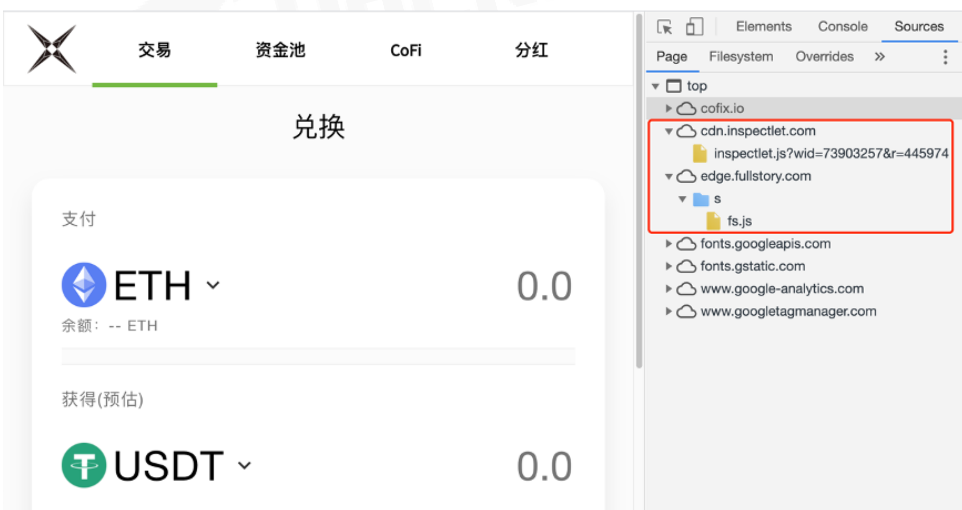

Figure 2.2: Invoked JS files

```
banner.page.html
coin-input.page.html
connect-wallet.page.html
coin-select.page.html
footer.page.html
header.page.html
liquid-select.page.html
liquid-input.page.html
profit.page.html
sider-menu.page.html
switch-lang.page.html
income.page.html
add-liquid.page.html
explain-liquid.page.html
redeem-liquid.page.html
warning-detail.page.html
warning-liquid.page.html
liquid.page.html
swap.page.html
app.component.html
```

Figure 2.3: HTML Template Files

```
<div class="banner-content">
  <div class="content">
    <h2 class="title">{{bannerConent.title | translate}}</h2>
    <p class="desc" *ngFor="let desc of bannerConent.descriptions;let i=index">
      {{desc | translate}}
    </p>
    <div class="more">
      <a (click)="goto(bannerConent.more.url)"
        >{{bannerConent.more.text | translate}}
      </a>
    </div>
  </div>
</div>
```

Figure 2.4: banner.page.html Template

Figure 2.5: banner.page.html Rendered Content

example, the variable values are hardcoded in resource files and cannot be changed during rendering. Therefore, the current implementation is not vulnerable to XSS vulnerabilities, as shown in Figure 2.6. We further checked all available template files and confirmed no XSS vulnerability exist in the given source code.

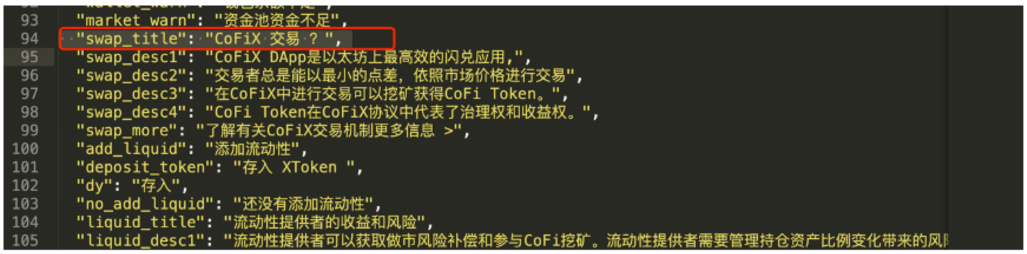

Figure 2.6: Hardcoded Variable Values

2.4 Inadequate HTTP Security Headers

By using Burp Suite, we quickly identified the inadequate HTTP headers usage of target system. As shown in Figure 2.7, the target system is hosted on Amazon S3 buckets statically and the JS files are served by CloudFront. Such mechanisms provides strong security and mitigates the cache poison risk, as long as the S3 deployment token is not leaked and kept with caution. However, a few HTTP Headers can be configured to enhance the security, as listed in Table 2.1. To add the missed HTTP security headers, the client can refer to Amazon Security Blog Post and leverage Lambda service [1].

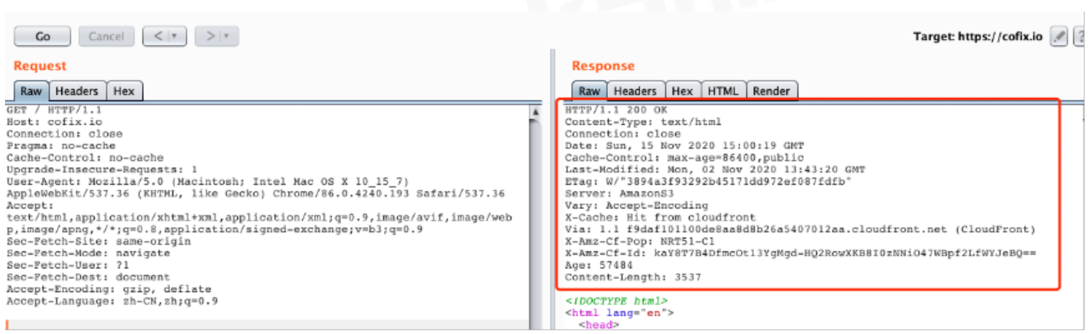

Figure 2.7: Presented HTTP Headers

2.5 Inadequate Cross-Origin Resource Sharing (CORS) Settings

Currently, there is no Access-Control-Allow-Origin or Access-Control-Allow-Credentials presented in response headers of the target system. As a result, there shouldn't be any CORS security issue at

Table 2.1: Missed Security HTTP Headers

HTTP Header
Strict-Transport-Security
X-Frame-Options
X-XSS-Protection
X-Content-Type-Options
Content-Security-Policy

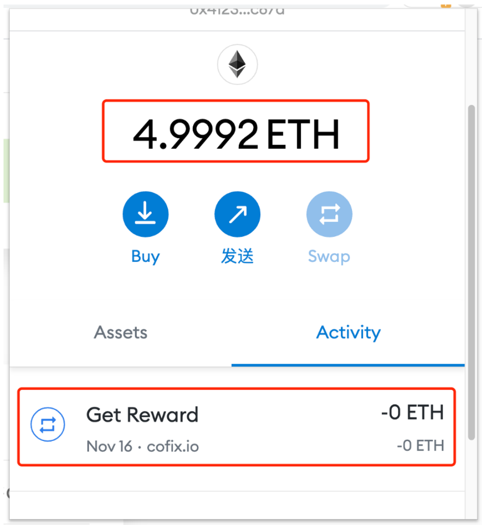

Figure 2.8: Reward Lost

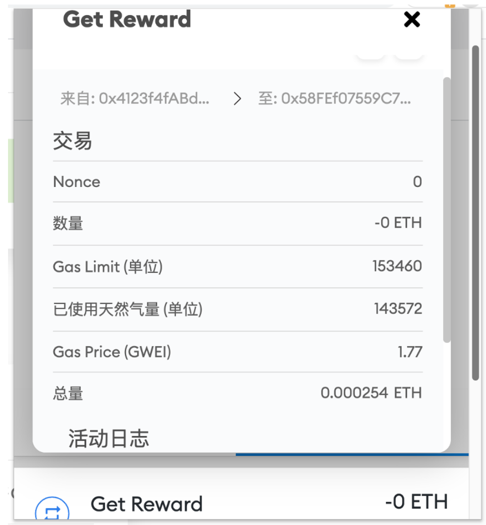

Figure 2.9: Reward Lost

this moment. However, the client should keep the current settings unless CORS is needed in the future.

2.6 Wallet Connection Security

While testing the target system, we identify an UI issue which could be reproduced by using MetaMask on ropsten chain. In particular, we deposit 5 ropsten Ether for testing. As shown in Figure 2.10, the frontend web UI fails to verify whether the amount of mined CoFi is 0 but proceed to invoke the smart contract to withdraw the fund. Such activity eventually lead to a waste of fee as shown in Figure 2.8 and Figure 2.9.

Figure 2.10: Missing Mined Token Verification

2.7 Insecure Session Management

As shown in Figure 2.11, there's no session management involved in target frontend web system.

Figure 2.11: No Session Presented

2.8 Cache Manipulation and Poisoning

As mentioned above, our client leverages Amazon S3 buckets and CloudFront to deliver their JS content and employed HTTPS policy with Amazon. As long as the S3 deployment token is kept secure, there is no need to worry about cache manipulation and poisoning.

2.9 Authentication and Authorization Bypass

There is no authentication mechanism implemented in the target system.

3 | Conclusion

During this white box code audit, we find our client shows excellent programming skills. The target under audit is designed with simple but secure way. There was no major security issue addressed in this code audit. Certain minor security issues can be fixed with minium effort, for instance, adding HTTP security headers with Amazon Lambda. We thank the prompt assistance from CoFix and their transparent code policy. We sincerely hope CoFix can have great achievement with their product.

Reference

[1] Amazon. Adding http security headers using lambda edge and amazon cloudfront. https://aws.amazon.com/blogs/networking-and-content-delivery/adding-http-security-headers-using-lambdaedge-and-amazon-cloudfront/, 2017.

[2] PeckShield. PeckShield Inc. https://www.peckshield.com.