# Dropbit

# Vulnerability Assessment

# Report

CERTIK

| | |
|---|---|
| Report ID | AS/DROPBIT- 01/APP |
| Report Title | Dropbit Vulnerability Report |
| Revision Number | I |
| Report Publish Date | 2019-03-03 |

## TABLE OF CONTENTS

## Executive Summary

3 High level, 3 Medium level, and 0 Low level vulnerabilities discovered.

| S.No. | Priority | Bug Title |
|---|---|---|
| 1. | Medium | Activity not protected |
| 2. | Medium | Salt hardcoded on client side |
| 3. | Medium | Cross site scripting (reflected) |
| 4. | High | Cross site scripting (reflected) |
| 5. | High | Cross site scripting (reflected) |
| 6. | High | SSL pinning missing on Android and iOS app |

## Confidentiality & Proprietary

## Background

Certik has completed security testing for Dropbit of their web/mobile application. The assessment was carried out using both a manual and automated approach.

## Report Analysis

1. The issues identified and proposed action plans in this report are based on our testing performed within the limited timespan and limited access to the servers. We made specific efforts to verify the accuracy and authenticity of the information gathered only in those cases where it was deemed necessary.

2. The identification of the issues in the report is mainly based on the tests carried out during the limited time for conducting such an exercise. As the basis of selecting the most appropriate weaknesses / vulnerabilities is purely judgmental in view of the time available, the outcome of the analysis may not be exhaustive and representing all possibilities, though we have taken reasonable care to cover the major eventualities.

3. The vulnerabilities reported in this report are **valid as of March 03, 2019.** Any vulnerability which may have been discovered after this or any exploit been made available after this period does not come under the purview of this report.

4. Any configuration changes or software / hardware updates made on hosts/machines or on the application covered in this test after the date mentioned herein may impact the security posture either positively or negatively and hence invalidates the claims & observations in this report. Whenever there is a change on the architecture, we recommend that you conduct vulnerability assessment and penetration test to ensure that your security posture is compliant with your security policies.

## Scope

Prior to commencing the assessment, Dropbit shared the scope with CertiK as mentioned below. Out of the overall scope, this report has been articulated focusing on the scope details mentioned below.

| Domain | Application name | Tier |
|---|---|---|
| api.coinninja.com | **Dropbit Android/iOS application** | **I** |

## Ports

| Port | State |
|---|---|
| **443** | **Open** |

**Threat Distribution**

There are three parameters considered for calculating the Inherent Risk Rating for a vulnerability, namely,

- CVSS rating for infrastructure elements based on its exposure
- Financial risk
- Customer impact of the application.

The parameters details are mentioned below.

- **CVSS Rating for Infrastructure Elements**

| Severity Level | Color Indicator | CVSS Category (Internet) | Severity impact risk score |
|---|---|---|---|
| **High** | Red | 8.00 – 10.00 | 60 |
| **Medium** | Orange | 4.00-7.99 | 40 |
| **Low** | Green | 0.01-3.99 | 20 |

## Vulnerabilities Summary

3 High level, 3 Medium level, and 0 Low level vulnerabilities discovered.

| S.No. | Priority | Bug Title |
|---|---|---|
| 1. | Medium | Activity not protected |
| 2. | Medium | Salt hardcoded on client side |
| 3. | Medium | Cross site scripting (reflected) |
| 4. | High | Cross site scripting (reflected) |
| 5. | High | Cross site scripting (reflected) |
| 6. | High | SSL pinning missing on Android and iOS app |

**Vulnerabilities and Recommendation:**

| 1. Activity not protected | |
|---|---|
| **Inherent Risk Rating** | Medium |
| **Affected Component** | **Dropbit Android** |
| | |
| **Description** | |
| An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Activity is explicitly exported.<br><br>com.coinninja.coinkeeper.view.activity.VerifyPhoneNumberActivity) is not Protected. | |
| **Vulnerable Activity** | |
| com.coinninja.coinkeeper.view.activity.VerifyPhoneNumberActivity | |
| | |
| **Fix Recommendation** | |
| android:exported : This element sets whether the activity can be launched by components of other applications — "true" if it can be, and "false" if not. If "false", the activity can be launched only by components of the same application or applications with the same user ID. If you are using intent filters, you should not set this element "false". If you do so, and an app tries to call the activity, system throws an ActivityNotFoundException. Instead, you should prevent other apps from calling the activity by not setting intent filters for it.<br><br>If you do not have intent filters, the default value for this element is "false". If you set the element "true", the activity is accessible to any app that knows its exact class name, but does not resolve when the system tries to match an implicit intent.<br><br>This attribute is not the only way to limit an activity's exposure to other applications. You can also use a permission to limit the external entities that can invoke the activity (see the permission attribute). | |

## 2. Salt hardcoded on client side

| | |
|---|---|
| **Inherent Risk Rating** | Medium |
| **Affected Component** | **Dropbit Android/iOS app** |

**Description**

During source code review we discovered that a few files have keys which are hardcoded in the code.

Please review them to see if they possess any risk or not.

**Vulnerable Code**

```
public final class BuildConfig {
    public static final String ANALYTICS_TOKEN = "1331ef0b555ee0ef48bf700035156337";
    public static final String APPLICATION_ID = "com.coinninja.coinkeeper";
    public static final int AUTHENTICATION_TIME_OUT_MS = 30000;
    public static final String BUILD_TYPE = "release";
    public static final boolean CN_DB_ENCRYPTION_ENABLED = true;
    public static final String COIN_NINJA_API_BASE = "https://api.coinninja.com";
    public static final boolean DEBUG = false;
    public static final String DEFAULT_SALT =
"2116F1A5-7B3A-4D10-928F-53F5D68E1878";
    public static final long DUST_AMOUNT_SATOSHIS = 500L;
    public static final String FCM_APPLICATION_KEY = "dropbit-prod-01";
    public static final String FLAVOR = "production";
    public static final int NUM_ADDRESSES_AHEAD = 10;
    public static final int VERSION_CODE = 10414;
    public static final String VERSION_NAME = "1.3.0";
```

**Fix Recommendation**

Remove hardcoded secrets from the client side.

| 3. Cross site scripting (reflected) | |
|---|---|
| **Inherent Risk Rating** | Medium |
| **Affected Component** | **coinninja.com** |
| **Description** | |

Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application.

The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes.

Users can be induced to issue the attacker's crafted request in various ways. For example, the attacker can send a victim a link containing a malicious URL in an email or instant message. They can submit the link to popular web sites that allow content authoring, for example in blog comments. And they can create an innocuous looking web site that causes anyone viewing it to make arbitrary cross-domain requests to the vulnerable application (using either the GET or the POST method).

The security impact of cross-site scripting vulnerabilities is dependent upon the nature of the vulnerable application, the kinds of data and functionality that it contains, and the other applications that belong to the same domain and organization. If the application is used only to display non-sensitive public content, with no authentication or access control functionality, then a cross-site scripting flaw may be considered low risk. However, if the same application resides on a domain that can access cookies for other more security-critical applications, then the vulnerability could be used to attack those other applications, and so may be considered high risk. Similarly, if the organization that owns the application is a likely target for phishing attacks, then the vulnerability could be leveraged to lend credibility to such attacks, by injecting Trojan functionality into the vulnerable application and exploiting users' trust in the organization in order to capture credentials for other applications that it owns. In many kinds of application, such as those providing online banking functionality, cross-site scripting should always be considered high risk

The value of the **q** request parameter is copied into the value of an HTML tag attribute which is encapsulated in double quotation marks. The payload **hv2gr"><a>w9lfr** was submitted in the q parameter. This input was echoed unmodified in the application's response.

This behavior demonstrates that it is possible to inject new HTML tags into the

| |
|---|
| returned document. An attempt was made to identify a full proof-of-concept attack for injecting arbitrary JavaScript but this was not successful. You should manually examine the application's behavior and attempt to identify any unusual input validation or other obstacles that may be in place.<br><br>The application appears to be blocking the usual proof-of-concept test string used by Burp, so an alternate test string was used. |
| |
| **Vulnerable URl** |
| https://coinninja.com/news/where-can-i-spend-my-bitcoin?q=Ddddhv2gr%22%3e%3ca%3ew9lfr |
| **Fix Recommendation** |
| In most situations where user-controllable data is copied into application responses, cross-site scripting attacks can be prevented using two layers of defenses:<br><br>• Input should be validated as strictly as possible on arrival, given the kind of content that it is expected to contain. For example, personal names should consist of alphabetical and a small range of typographical characters, and be relatively short; a year of birth should consist of exactly four numerals; email addresses should match a well-defined regular expression. Input which fails the validation should be rejected, not sanitized.<br>• User input should be HTML-encoded at any point where it is copied into application responses. All HTML metacharacters, including < > " ' and =, should be replaced with the corresponding HTML entities (&lt; &gt; etc).<br><br>In cases where the application's functionality allows users to author content using a restricted subset of HTML tags and attributes (for example, blog comments which allow limited formatting and linking), it is necessary to parse the supplied HTML to validate that it does not use any dangerous syntax; this is a non-trivial task. |

| 4. Cross site scripting (reflected) | |
|---|---|
| **Inherent Risk Rating** | High |
| **Affected Component** | **coinninja.com** |
| | |
| **Description** | |

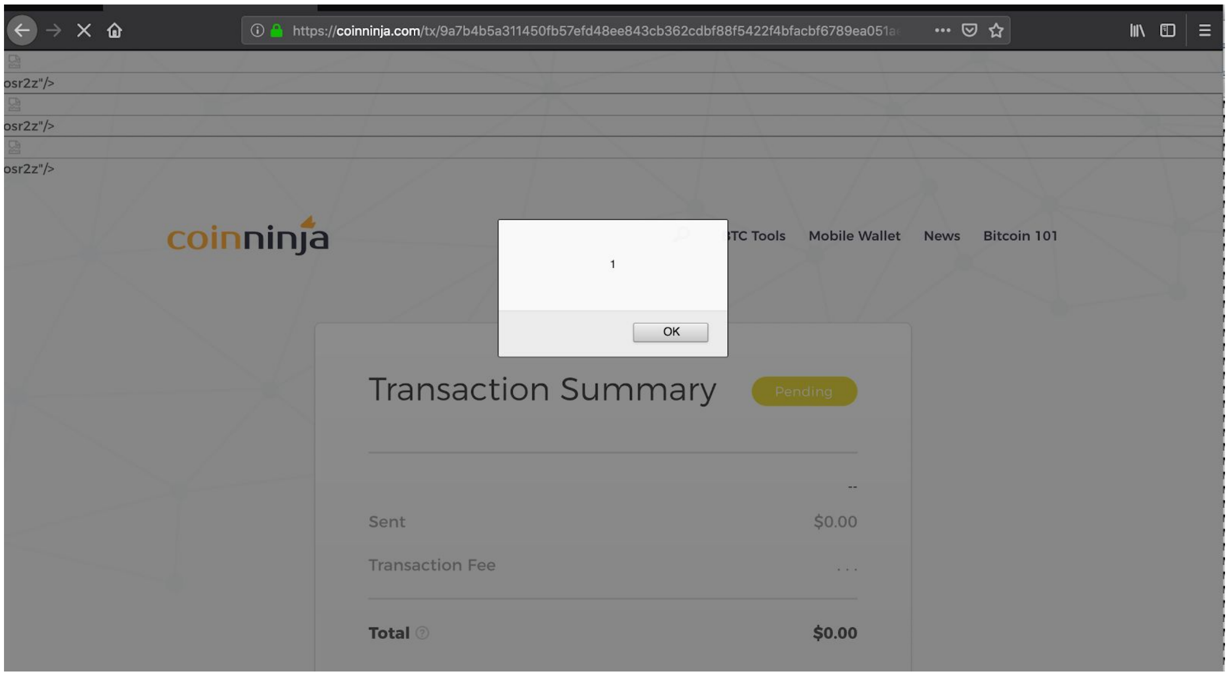
Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application.
The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes.
Users can be induced to issue the attacker's crafted request in various ways. For example, the attacker can send a victim a link containing a malicious URL in an email or instant message. They can submit the link to popular web sites that allow content authoring, for example in blog comments. And they can create an innocuous looking web site that causes anyone viewing it to make arbitrary cross-domain requests to the vulnerable application (using either the GET or the POST method).
The security impact of cross-site scripting vulnerabilities is dependent upon the nature of the vulnerable application, the kinds of data and functionality that it contains, and the other applications that belong to the same domain and organization. If the application is used only to display non-sensitive public content, with no authentication or access control functionality, then a cross-site scripting flaw may be considered low risk. However, if the same application resides on a domain that can access cookies for other more security-critical applications, then the vulnerability could be used to attack those other applications, and so may be considered high risk. Similarly, if the organization that owns the application is a likely target for phishing attacks, then the vulnerability could be leveraged to lend credibility to such attacks, by injecting Trojan functionality into the vulnerable application and exploiting users' trust in the organization in order to capture credentials for other applications that it owns. In many kinds of application, such as those providing online banking functionality, cross-site scripting should always be considered high risk.

The value of the URL path filename is copied into the value of an HTML tag attribute which is encapsulated in double quotation marks. The payload **p7emn"><img src=a onerror=alert(1)>osr2z** was submitted in the URL path filename. This input was echoed unmodified in the application's response.

This proof-of-concept attack demonstrates that it is possible to inject arbitrary JavaScript into the application's response. The proof-of-concept attack demonstrated uses an event handler to introduce arbitrary JavaScript into the document.

**Vulnerable URl**

https://coinninja.com/tx/9a7b4b5a311450fb57efd48ee843cb362cdbf88f5422f4bfacbf6789ea051ae2p7emn%22%3E%3Cimg%20src%3da%20onerror%3dalert(1)%3Eosr2z



**Fix Recommendation**

In most situations where user-controllable data is copied into application responses, cross-site scripting attacks can be prevented using two layers of defenses:

- Input should be validated as strictly as possible on arrival, given the kind of content that it is expected to contain. For example, personal names should consist of alphabetical and a small range of typographical characters, and be relatively short; a year of birth should consist of exactly four numerals; email addresses should match a well-defined regular expression. Input which fails the validation should be rejected, not sanitized.
- User input should be HTML-encoded at any point where it is copied into application responses. All HTML metacharacters, including < > " ' and =, should be replaced with the corresponding HTML entities (&lt; &gt; etc).

In cases where the application's functionality allows users to author content using a restricted subset of HTML tags and attributes (for example, blog comments which allow limited formatting and linking), it is necessary to parse the supplied HTML to validate that it does not use any dangerous syntax; this is a non-trivial task.

## 5. Cross site scripting (reflected)

| | |
|---|---|
| **Inherent Risk Rating** | High |
| **Affected Component** | coinninja.com |
| | |
| **Description** | |

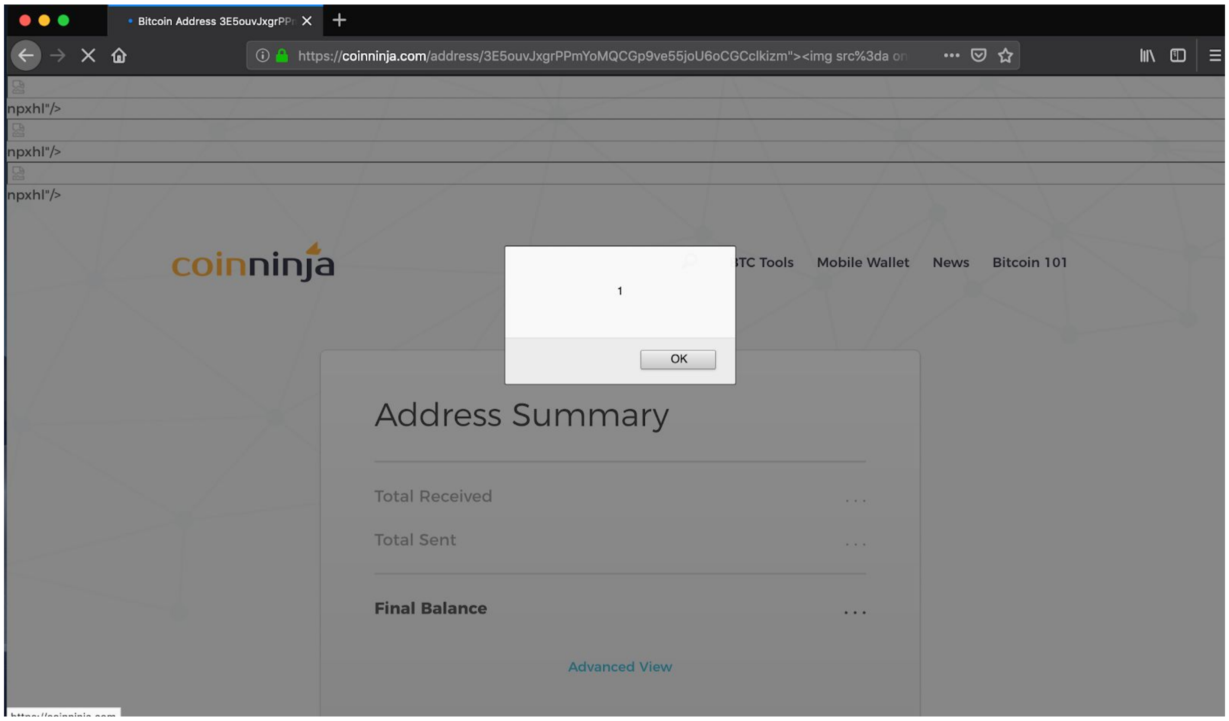
Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application.
The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes.
Users can be induced to issue the attacker's crafted request in various ways. For example, the attacker can send a victim a link containing a malicious URL in an email or instant message. They can submit the link to popular web sites that allow content authoring, for example in blog comments. And they can create an innocuous looking web site that causes anyone viewing it to make arbitrary cross-domain requests to the vulnerable application (using either the GET or the POST method).
The security impact of cross-site scripting vulnerabilities is dependent upon the nature of the vulnerable application, the kinds of data and functionality that it contains, and the other applications that belong to the same domain and organization. If the application is used only to display non-sensitive public content, with no authentication or access control functionality, then a cross-site scripting flaw may be considered low risk. However, if the same application resides on a domain that can access cookies for other more security-critical applications, then the vulnerability could be used to attack those other applications, and so may be considered high risk. Similarly, if the organization that owns the application is a likely target for phishing attacks, then the vulnerability could be leveraged to lend credibility to such attacks, by injecting Trojan functionality into the vulnerable application and exploiting users' trust in the organization in order to capture credentials for other applications that it owns. In many kinds of application, such as those providing online banking functionality, cross-site scripting should always be considered high risk.

The value of the URL path filename is copied into the value of an HTML tag attribute which is encapsulated in double quotation marks. The payload **lkizm"><img src=a onerror=alert(1)>npxhl** was submitted in the URL path filename. This input was echoed unmodified in the application's response.

This proof-of-concept attack demonstrates that it is possible to inject arbitrary JavaScript into the application's response. The proof-of-concept attack demonstrated uses an event handler to introduce arbitrary JavaScript into the document.

**Vulnerable URl**

https://coinninja.com/address/3E5ouvJxgrPPmYoMQCGp9ve55joU6oCGCclkizm%22%3e%3cimg%20src%3da%20onerror%3dalert(1)%3enpxhl



**Fix Recommendation**

In most situations where user-controllable data is copied into application responses, cross-site scripting attacks can be prevented using two layers of defenses:

- Input should be validated as strictly as possible on arrival, given the kind of content that it is expected to contain. For example, personal names should consist of alphabetical and a small range of typographical characters, and be relatively short; a year of birth should consist of exactly four numerals; email addresses should match a well-defined regular expression. Input which fails the validation should be rejected, not sanitized.
- User input should be HTML-encoded at any point where it is copied into application responses. All HTML metacharacters, including < > " ' and =, should be replaced with the corresponding HTML entities (&lt; &gt; etc).

| |
|---|
| In cases where the application's functionality allows users to author content using a restricted subset of HTML tags and attributes (for example, blog comments which allow limited formatting and linking), it is necessary to parse the supplied HTML to validate that it does not use any dangerous syntax; this is a non-trivial task. |

| 6. SSL pinning missing on Android/iOS app | |
|---|---|
| **Inherent Risk Rating** | High |
| **Affected Component** | Android/iOS |
| | |
| **Description** | |
| SSL Pinning is making sure the client checks the server's certificate against a known copy of that certificate. Simply bundle your server's SSL certificate inside your application, and make sure any SSL request first validates that the server's certificate exactly matches the bundle's certificate.<br><br>iOS SSL pinning:<br>https://possiblemobile.com/2013/03/ssl-pinning-for-increased-app-security/<br>Android SSL pinning:<br>https://medium.com/@appmattus/android-security-ssl-pinning-1db8acb6621e | |
| | |
| **Vulnerable URl** | |
| **Fix Recommendation** | |
| Implement SSL pinning in both Android and iOS application. | |