Zellic

LootRush

Custodial Wallet Security Assessment

July 10, 2023

Prepared for:

LootRush

Prepared by:

Jasraj Bedi and Maik Robert

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for LootRush from July 3rd to July 6th, 2023. During this engagement, Zellic reviewed LootRush's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any vulnerabilities that would let a user have full control over assets in the custodial wallet?
- Is it possible for a user to manipulate the system in a way that falsely indicates an asset has been returned without actually returning it?
- Are there any vulnerabilities that would allow a user to abuse the wallet functionality in unintended ways?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Interactions with other parts of the codebase not in scope for the audit
- Front-end components
- Infrastructure relating to the project

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped LootRush files, we discovered one finding, which was of medium impact.

Additionally, Zellic recorded its notes and observations from the assessment for LootRush's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	0
Informational	0

2 Introduction

2.1 About LootRush

LootRush, a custodial wallet product, is based on the MetaMask extension and aims to enable users to rent assets at www.lootrush.com. The objective is to provide users with the ability to sign in to games and play with rented assets without the risk of losing them, such as unauthorized selling or transferring.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in web-based applications arise from oversights during development. Missing an authentication check on a single API endpoint can circumvent the entire authentication model of the application. Failure to properly sanitize and encode user input can lead to vulnerabilities like SQL injection or cross-site scripting. We use automated tools to identify unsafe code patterns and perform a thorough manual review for vulnerable code patterns.

Business logic errors. Business logic is the heart of all applications. We manually review logic to ensure that the code implements the expected functionality specified in the platform's design documents. We also thoroughly examine the specifications and designs for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse.

Integration risks. Web projects contain third-party dependencies and interact with APIs and code that are not under the developer's control. Auditors will review the project's external interactions and identify potentially dangerous behavior, such as implicitly trusting API responses or assuming third-party code functionality.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also suggest possible improvements to code clarity, documentation, and usability.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

LootRush Files

Repositories	https://github.com/smelthq/player-one/
	https://github.com/smelthq/lootrush-extension/
Versions	player-one: `dbab699c36028f4b30071e34bc957f495c2a1019`
	lootrush-extension: `c9d58294fcc02a00163c1ec5317b71f9c471048f`
Programs	• packages/rest-apis/wallet-api/src/app/*
	• lootrush/lib/LootrushKeyring.js
Types	TypeScript, JavaScript
Platforms	Windows, MacOS, Linux

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of one person-week. The assessment was conducted over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Maik Robert, Engineer
maik@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 30, 2023	Kick-off call
July 3, 2023	Start of primary review period
July 6, 2023	End of primary review period

3 Detailed Findings

3.1 Insecure default value for JWT secret

- **Target**: packages/rest-apis/wallet-api/src/app/config.ts
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The custodial wallet uses JSON Web Tokens (JWTs) to authenticate a user to the back-end. To prevent a malicious user from tampering with the contents of the JWTs, a secret only known to the server is used to sign the token.

```
import * as env from 'env-var'

export const ENV_NAME = env.get('ENV_NAME').required().asString()

export const BROKER_URL = env.get('BROKER_URL').required().asUrlString()
export const HTTP_PORT = env.get('HTTP_PORT').required().asPortNumber()
export const SERVICE_NAME = env.get('K_SERVICE').required().asString()
export const CLIENT_JWT_SECRET = env
  .get('CLIENT_JWT_SECRET')
  .default('sekret')
  .asString();
export const WALLET_API_KMS_ID
    = env.get('WALLET_API_KMS_ID').required().asString()
export const INFURA_API_KEY
    = env.get('INFURA_API_KEY').required().asString()
export const STATSIG_SERVER_KEY
    = env.get('STATSIG_SERVER_KEY').required().asString()
```

The code uses the `env-var` package to dynamically pull secrets from environment variables. In case the `CLIENT_JWT_SECRET` is not set, it will default to an easily guessable string — `sekret`.

Impact

An easily guessed JWT secret would allow an attacker to sign tokens with arbitrary data, for example an admin email address. A valid JWT with an admin email address would let the attacker call any function with any parameters, thus affecting all custodial wallets.

Recommendations

In an ideal case, the `.required()` method should be used; this forces the environment variable to be present and would throw an error if it is missing. If a default fallback value has to be used, it should be replaced with a strong, randomly generated string that is highly secure and cannot be guessed or bruteforced.

Remediation

This issue has been acknowledged by LootRush, and a fix was implemented in commit a90348f9.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 General testing methodology

The scope for this audit was sufficiently small to focus heavily on manual review of the code. Two potential impacts have been outlined as the main focus for this test, these being the ability for a user to have unintended full control over assets in the custodial wallet and the ability for a user to trick the return system into thinking an asset has been returned, while still being in possession of the asset.

With these goals in mind, Zellic tried to identify general web vulnerabilities as well as logical bugs that would allow a user to achieve that. Given below are a few examples of tests and potential logical bugs that Zellic investigated; this list is not comprehensive.

Query tampering

As the returning of assets involves a database, the most immediate way an attacker could potentially trick the application into thinking an asset has been returned is by tampering with the queries.

While attacker-controlled input is passed into the queries, it is done in a safe way. For example,

```
class ReturnOrderItemsController {
  async create(req, res) {
    const { userId } = req?.auth?.isAdmin ? req.params : req.auth;
    const { rentalOrderItemIds = [] } = req.body;
    (...)
    const returnableAssets
    = await orderService.fetchReturnableAssets(userId,
    rentalOrderItemIds)
    (...)
  }
}
```

Source: packages/rest-apis/wallet-api/src/app/controllers/ReturnOrderItemsController.ts

here the user has partial control over the `userId` and full control over the `rentalOrderItemIds`. These are then passed to the `orderService.fetchReturnableAssets()` function.

```
async fetchReturnableAssets(userId: string, rentalOrderItemIds: string[])
    {
    const itemsToReturn = await lootrushDbReplica(
      'marketplace.rental_orders_items as roi'
    )
      .select(
        'roi.id',
        ( ... )
      )
      ( ... )
      .where({
        userId,
      })
      .whereIn('roi.id', rentalOrderItemIds)
      .whereNot('roi.status', RentalOrderItemStatus.RETURNED);
    if (itemsToReturn.length > 0) {
      return itemsToReturn;
    }

    return [];
  }
```

Source: packages/rest-apis/wallet-api/src/app/services/RentalOrderService.ts

Here the user-controlled input is passed into the SQL queries in a safe manner by using the Knex query builder. The same pattern is used across the audited code. If, for example, the input would have been passed into Knex's `.raw()` query, a user may be able to manipulate the input in a way that would result in running arbitrary SQL commands, tricking the return logic.

Asset returns

Another potential way a user could trick the system is by returning someone elses assets. This type of logical bug is prevented in multiple ways.

First, the JWT uniquely identifies the user and checks if the user is an admin. A middleware is used to verify the JWT before every controller method invocation. This ensures that the JWT which the user has provided has not been tampered with.

When it is then accessed in the controller, a check is performed using the JWT if the user is an admin. If the user is not an admin, only values that are contained inside the JWT are used for processing, preventing a malicious actor from passing arbitrary values to functions.

```
class ReturnOrderItemsController {
  async create(req, res) {
    const { userId } = req?.auth?.isAdmin ? req.params : req.auth;
    ( ... )
  }
}
```

Source: packages/rest-apis/wallet-api/src/app/controllers/ReturnOrderItemsController.ts

Further, every action that is taken is done so using the `userId`, which is always constructed using the JWT. This ensures that actions are always for the intended user.

```
class ReturnOrderItemsController {
  async create(req, res) {
    const { userId } = req?.auth?.isAdmin ? req.params : req.auth;
    ( ... )
    const returnableAssets
    = await orderService.fetchReturnableAssets(userId,
    rentalOrderItemIds)
    ( ... )
    const wallet = await custodialWalletService.find({ userId }, false);
    const cryptoWallet
    = await custodialWalletService.getCryptoWalletInstance(wallet)
    const result = await orderService.returnAssets(returnableAssets,
    cryptoWallet);
    return res.status(200).send({ result })
  }
}
```

Source: packages/rest-apis/wallet-api/src/app/controllers/ReturnOrderItemsController.ts

It should be noted that while Zellic did not identify issues with the handling of JWTs inside the audited code, interactions or handling of JWTs outside the audited code may impact the behavior in unintended ways.

Method firewall

In combination with the above mentioned mitigations, the code also makes use of a method firewall, which restricts the methods a user is allowed to call for their custodial wallet.

When a user requests to call a method, the request is routed through the controller, which checks the JWT, as demonstrated above, to make sure that the user does not call methods on other wallets but their own. The request then ends up in the `callMethod` handler.

```
async callMethod(
    wallet: UserCustodialWallet,
    method: string,
    params
  ): Promise<string | boolean> {
    const cryptoWallet = await this.getCryptoWalletInstance(wallet);
    const isMethodAllowedFN = methodFirewall[method]
    || methodFirewall.default;
    const isMethodAllowed = await isMethodAllowedFN(params);
    const [walletCall] = await lootrushDb<UserCustodialWalletCall>(
      'user_custodial_wallet_calls'
    )
      .insert({
        id: uuid(),
        userCustodialWalletId: wallet.id,
        method,
        params: JSON.stringify(params),
        response: JSON.stringify({ isMethodAllowed }),
      })
      .returning('*');

    // verify if transaction can procceed
    if (!isMethodAllowed) return '';

    const response = await cryptoWallet.callMethod(method, params);
    await
    lootrushDb<UserCustodialWalletCall>('user_custodial_wallet_calls')
      .where({
        id: walletCall.id,
      })
      .update({
        response: JSON.stringify(response),
      });

    return response;
  }
```

Source: packages/rest-apis/wallet-api/src/app/services/CustodialWalletService.ts

Here the user's wallet is retrieved, the method that has been passed is checked against the allowlist, and then it is either rejected or executed depending on if the method was in the allowlist. During the audit, Zellic did not discover any instances where an attacker could circumvent the logic implemented in the audited code.

4.2 Admin model

Given the custodial nature of the wallet, the team needs a way to take administrative action on user wallets. While Zellic did not identify any vulnerabilities in the audited section of the application regarding this functionality, we feel that the current implementation constitutes a single point of failure and a likely attack target for malicious actors.

Every request is routed through a middleware that verifies the JWT passed with the request.

```
export const AuthMiddleware = async (req, res, next) ⇒ {
  const { token = {} } = req.cookies;
  const { authorization } = req.headers;
  const authToken = authorization?.split('Bearer ')[1];

  if (!authToken && !token) return res.status(401).send({ error:
    'Unauthorized' });

  try {
    const decodedToken = jwt.verify(
      authToken || token,
      CLIENT_JWT_SECRET
    ) as jwt.JwtPayload;
    req.auth = {
      userId: decodedToken.sub,
      sessionId: decodedToken.jti,
      isAdmin: ADMIN_EMAIL_LIST.includes(decodedToken.email),
      email: decodedToken.email
    };
  } catch (e) {
    return res
      .status(401)
      .send({ error: `JWT could not be verified due to ${e.message}` });
  }

  return next();
};
```

Source: packages/rest-apis/wallet-api/src/app/middlewares/AuthMiddleware.ts

Here the `req.auth` object is constructed. One of the fields is the `isAdmin` field, which is a simple check if the supplied user email is contained in a list of admin emails. The list contains 14 @lootrush.com employee emails.

The controllers then access this auth object and check if the `isAdmin` field is true. If it is false, meaning the user is not an admin, the values from the auth object are used that have been signed in the JWT. If the user is an admin, then the parameters are pulled from the request, allowing an admin to interact with wallets not belonging to them.

```
class CustodialWalletMethodCallsController {
  async create(req, res) {
    try {
      const { userId } = req?.auth?.isAdmin ? req.params : req.auth;
      (...)
      const wallet = await walletService.find({ userId, address:
    callPayload.walletAddress }, false);
      if (!wallet)
        return res.status(404).send();
      const result = await walletService.callMethod(wallet,
    callPayload.method, callPayload.params);
      return res.status(200).json({ ...callPayload, result });
    } (..)
  }
}
```

Source: packages/rest-apis/wallet-api/src/app/controllers/CustodialWalletMethodsController.ts

We believe this will be a primary target for attackers as it allows almost total control of the custodial system. A compromise of a single admin email could compromise the entire application.

Further, there may be vulnerabilities in the rest of the codebase that would allow an attacker to forge JWT tokens with arbitrary emails, allow account takeover issues on the website itself, or allow email parsing issues, which would all lead to the same impact of an admin account being used to compromise the entire system as a result.

A more robust solution may be found that prevents an admin account from taking direct actions on user wallets, or something like a multi-signature approach requiring multiple admin approvals to take direct actions on user wallets.

5 Audit Results

At the time of our audit, the audited code was not in use in production.

During our assessment on the scoped LootRush files, we discovered one finding, which was of medium impact. LootRush acknowledged the finding and implemented a fix.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.