TYMIO Security Analysis

by Pessimistic

This report is public

March 13, 2024

Abstract

In this report, we consider the security of smart contracts of TYMIO project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. A single audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, a security audit is not investment advice.

Summary

In this report, we considered the security of TYMIO smart contracts. We described the audit process in the section below.

The initial audit showed several issues of medium severity: Documentation, Lack of approval to router, Potential risks to the user's deposit from the project owners, Project roles, Token transfers. Also, several low-severity issues were found.

After the initial audit, the codebase was updated. Most of the issues from the initial audit were fixed. The severity of the Documentation issue was decreased from medium to low. Moreover, the Project roles issue was moved to notes. We verified that the user funds are safe (see N06). However, several new low-severity issues that need to be addressed were found. The newfound issues were fixed or commented on afterward.

The overall code quality is average. The code is covered by NatSpec comments. However, the code contains calculations that lack technical documentation.

General recommendations

We recommend fixing the mentioned issues, addressing newly identified issues, running the linter/prettifier on the code, and making the technical documentation publicly accessible. Moreover, we recommend testing on the locally forked mainnet.

Project overview

Project description

For the audit, we were provided with TYMIO project on a public GitHub repository, commit adf4dad7d90a78a2d6445aabca7cb394cf2c7893.

The scope of the audit included:

- **contracts/PayerV3.sol**.

The documentation for the project was not provided.

All 29 tests pass successfully. The code coverage is 100%.

The total LOC of audited sources is 383.

Codebase update

After the initial audit, the codebase was updated, and we were provided with commit c1256fb3b769cb62b9d960baef551dca6f44dfc0.

The developers fixed all the medium-severity issues. Along with that, minimal private documentation was added. Also, they fixed all the low-severity issues and addressed the notes. All 29 out of 29 tests passed successfully. The code coverage was 98.8%.

Audit process

We started the audit on February 27, 2024, and finished on March 1, 2024.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project. After a discussion, we performed preliminary research and started the review.

During the work, we stayed in touch with the developers and discussed confusing or suspicious parts of the code.

We manually analyzed the contract within the scope of the audit and checked its logic. Among other, we checked the following properties of the contract:

- Users can claim their tokens without any limitations if the off-chain client remains inactive;
- A single order should not disrupt the execution of the entire batch;
- Whether users' deposits are safe (see M02 and N06);
- Potential arithmetic issues.

We scanned the project with the following tools:

- Static analyzer Slither;
- Our plugin Slitherin with an extended set of rules;
- Semgrep rules for smart contracts. We also sent the results to the developers in the text file.

We ran tests and calculated the code coverage.

We combined in a private report all the verified issues we found during the manual audit or discovered by automated tools.

We made the recheck on March 11, 2024 and finished on March 12, 2024. We checked whether the developers fixed previous issues. We also ran the tests and calculated the code coverage. Finally, we updated the report.

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

The audit showed no critical issues.

Medium severity issues

Medium severity issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. Lack of approval to router (fixed)

In **PayerV3**, the UniswapV3 router is used for swaps but lacks token approval, leading to swap failures. (Note: the Uniswap router utilizes `transferFrom` function.)

The issue has been fixed and is not present in the latest version of the code.

M02. Potential risks to the user's deposit from the project owners (fixed)

There is no direct way for the project owners to withdraw users' deposits. However, currently, there are multiple indirect ways that allow project owners to influence user's deposit in some way in the **PayerV3** contract:

- The contract contains functionality that allows owners to withdraw the user's deposit, after `fullAccessAfter` seconds from the last user activity. According to specs, it will be 365 days. However, it is set as 1 second (according to the comment, for testing purposes). Moreover, the contract also uses some other testing values over production values. We recommend setting only the values/constants that will be used in the production..
- The owner can set `payerAddress`, from which all rewards are paid. Currently, there is no protection from setting this address to some user's address with a deposit and paying rewards from the user's balance.

The issues have been fixed and are not present in the latest version of the code.

M03. Token transfers (fixed)

In the current version of **PayerV3**, USDT will be used as an acceptable token for performing the swap. However, it's not possible to deposit this token because the contract expects the returned value on line 140, which will not be returned in the case of USDT. Moreover, at line 286, the return value is not checked. Consider using **SafeERC20** for token transfers in the **PayerV3** contract.

The issue has been fixed and is not present in the latest version of the code.

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. Duplicate token addition (fixed)

To add a token to the system, `onlyOwners` can utilize the `editAcceptableToken()` function. However, there are no validations in place to prevent the addition of tokens already included. This could lead to the same token being added to `acceptableTokensArray` multiple times, potentially causing unexpected behavior in `editAcceptableToken` function of **PayerV3** contract.

The issue has been fixed and is not present in the latest version of the code.

L02. Solidity version (fixed)

The solidity version in **PayerV3** is not fixed. Consider fixing the version to common minor (either 0.7.x or 0.8.x) and using latest version of the compiler.

The issue has been fixed and is not present in the latest version of the code.

L03. Unoptimized structure (fixed)

The order finalization process has a complexity of O(n^2), and as it involves numerous iterations over all token pairs to finalize orders, gas consumption could increase quadratically when new tokens are added. Additionally, there are opportunities to optimize gas consumption by caching state variables to memory or by resetting `swapsIn/swapsOut` to 1 wei instead of 0.

> *Comment from the developers: Code is optimized for gas usage. Caching state variables to memory and resetting swapsIn/swapsOut to 1, also some recommendations from `TYMIO_semgrep.txt` in our case are increasing the gas consumption, not reducing it.*

The issue has been fixed and is not present in the latest version of the code.

L04. Documentation

There is currently no technical documentation provided for **PayerV3**. We recommend focusing on developing the specification, particularly for the order settlement process.

The documentation is a critical part that helps to improve security and reduce risks. It should explicitly explain the purpose and behavior of the contracts, their interactions, and key design choices. It is also essential for any further integrations.

> *The developers prepared the initial version of the technical documentation. We recommend continuing work on the documentation.*

L05. Logical expressions (commented)

The codebase contains logical expressions, which have logical negate over full expression, as example:
`!(order.completed || block.timestamp > order.endTimestamp + maxExecutionTime)`.
These expressions are hard to read and comprehend. Moreover, the compiler may not optimize them, which will result in a redundant negate opcodes in the bytecode. Consider inversing the whole expression.

> *Comment from the developers:* *Won't fix.*

L06. User action time is not updated during the makeOrder (fixed)

User action time is not updated during the `makeOrder()`. Even though it is updated during `claimOrder()`, consider also calling `_updateUserActionTime` in the `makeOrder()` function.

The issue has been fixed at commit 8a2ae304a83bf6a34f57f45daffd983933a62858.

Notes

N01. Immutable vs State variables (fixed)

Several state variables might be turned into immutable in order to save gas in **PayerV3** contract.

The issues have been fixed and are not present in the latest version of the code.

N02. Possible DoS (fixed)

PayerV3 does not impose any limitations on the token amounts supplied during order creation. Consequently, this could create an opportunity for submitting numerous orders with only a fraction of the tokens being supplied. Consider implementing on-chain checks for minimum amount thresholds per each token in **PayerV3** contract.

The `minimalTokenAmounts` mapping has been added to define the threshold per token.

N03. Dynamic parameters (fixed)

Upon migrating `PayerV3`, a deployer is able to specify an arbitrary contract as a router. Along with that, the `wethAddress` used to deposit and withdraw eth is also assigned during the migration.

The `swapRouter` and `wethAddress` are now constantly assigned.

N04. Price discrepancy between the execution and simulation phases (commented)

Currently, an off-chain service is responsible for determining which orders should be swapped.

- When considering a buy order, the swap will only proceed if the price at the time of swapping exceeds `order.price`;
- If the sell order is considered, the swap will only proceed if the price at the time of swapping is lower than `order.price`.

However, it's important to note that the on-chain exchange rate during execution might not match that during simulation. This discrepancy can lead to underflow on line 228, followed by the entire batch being reverted in **PayerV3** contract.

> *Comment from the developers: Owners of the system have control over which order to swap. Orders with price discrepancies would not be swapped.*

N05. **Project roles** (fixed)

A functionality marked by `onlyOwners` includes:

- Initiating fund transfers for inactive users using the `emergencyQuit()` function;
- Executing orders with specified slippages;
- Withdrawing the dust or mistakenly sent ETH to the PayerV3 contract;
- Introducing new token pairs by invoking `editAcceptableToken()`;
- Changing the `payerAddress`;
- Changing the `poolFee`;
- Changing the `owner2`;
- Changing the `owner1`;
- Changing the `service`.

There are scenarios that can lead to undesirable consequences for the project and its users, e.g. if the private key for this role becomes compromised. We recommend designing contracts in a trustless manner or implementing proper key management, e.g., setting up a multisig.

<u>The issues have been fixed and are not present in the latest version of the code.</u>

> *<u>Comment from the developers:</u> In our architecture, admins do not have access to user funds or critical functions. We use two separate owners' wallets to ensure stability and uninterrupted access to system management; both admins can sign transactions independently. In the case of a 2/2 multisig, both owners would have to be online to sign, which is not suitable for our architecture since signing is time-sensitive.*

N06. **Safety of user deposits** (new)

After the proper initialization of the contract: setting `swapRouter`, `wethAddress`, and `fullAccessAfter`, to the values that are stated in the docs, all users' deposits are safe, and nobody, including the project owners, will not be able to take the user's deposit.

This analysis was performed by Pessimistic:

Yhtyyar Sahatov, Security Engineer
Rasul Yunusov, Security Engineer
Konstantin Zherebtsov, Business Development Lead
Irina Vikhareva, Project Manager
Alexander Seleznev, Founder

March 13, 2024