AUDITONE

Audit Report

payy

03.12.2024

Table of Contents

CONTENTS

01.
Project Description 3

02.
Project and Audit Information 4

03.
Contracts in scope 5

04.
Executive Summary 6

05.
Severity definitions 7

06.
Audit Overview 8

07.
Audit Findings 9

08.
Disclaimer 24

INTRODUCTION

Smart Contract Security Analysis Report

Note: This report may contain sensitive information on potential vulnerabilities and exploitation methods. This must be referred internally and should be only made available to the public after issues are resolved (to be confirmed prior by the client and AuditOne).

INTRODUCTION

Ubermensch3dot0, Ozoooneth and Eugenioclrc, who are auditors at AuditOne, successfully audited the smart contracts (as indicated below) of Payy-Rollup. The audit has been performed using manual analysis. This report presents all the findings regarding the audit performed on the customer's smart contracts. The report outlines how potential security risks are evaluated. Recommendations on quality assurance and security standards are provided in the report.

01-PROJECT DESCRIPTION

Payy is a platform designed to facilitate self-sovereign banking through cryptocurrency, offering a simplified and secure approach to the financial services traditionally managed by banks. The platform integrates a zk-rollup tailored for applications requiring privacy and scalability in payments and decentralized finance (DeFi).

The Payy Wallet is a non-custodial mobile wallet built around a stablecoin core, providing a user-friendly experience reminiscent of traditional banking apps but with the added benefits of cryptocurrency. This wallet supports private and instant money transfers using links, which streamline the payment process and reduce the complexity typically associated with blockchain transactions.

Additionally, Payy includes a feature called Payy Intents, a non-custodial proxy that simplifies the process of lending, market making, and staking on various DeFi protocols across multiple chains. This component allows users to engage in DeFi activities with greater ease compared to traditional methods, which often involve multiple steps and complex interactions with blockchain networks.

This audit focus on the updated version of the Roullup contract.

02-Project and Audit Information

Term	Description
Auditor	Ubermensch3dot0, Ozoooneth and Eugenioclrc
Reviewed by	Luis Buendia and Gracious Igwe
Type	Rollup
Language	Solidity
Ecosystem	EVM
Methods	Manual Review
Repository	https://github.com/polybase/zk-rollup/
Commit hash (at audit start)	0f76139001dff7d4d5d5ffbf9dc89389ec66cc17
Commit hash (after resolution)	cb86b728776d9d14451ed1f1b9a2523b13c2437d
Documentation	https://docs.payy.network/
Unit Testing	N/A
Website	https://payy.network/
Submission date	04/11/2024
Finishing date	28/11/2024

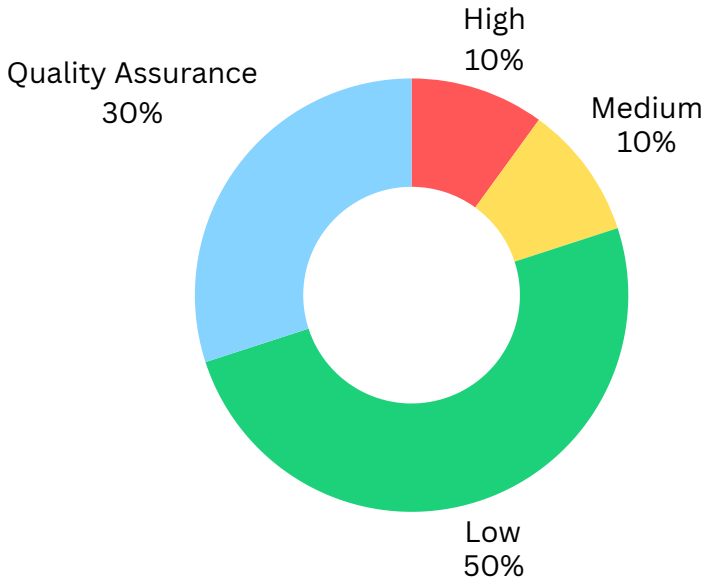
03-Contracts in Scope

Contract in scope:

- RollupV6.sol

04-Executive summary

Payy-Rollup smart contracts were audited between 04-11-2024 and 28-11-2024 by Ubermensch3dot0, Ozooooneth and Eugenioclrc. Manual analysis was carried out on the code base provided by the client. The following findings were reported to the client. For more details, refer to the findings section of the report.



Issue Category	Issues Found	Resolved	Acknowledged
High	1	1	0
Medium	1	1	0
Low	5	5	0
Quality Assurance	3	1	2

05-Severity Definitions

Risk factor matrix	Low	Medium	High
Occasional	L	M	H
Probable	L	M	H
Frequent	M	H	H

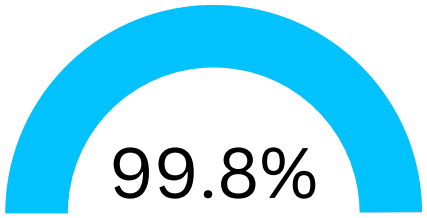
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

Medium: The impact of medium issues is less critical than high, but still probable with considerable damage. The protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions.

Low: Low issues impose a small risk on the project. Although the impact is not estimated to be significant, we recommend fixing them on a long-term horizon. Assets are not at risk: state handling, function incorrect as to spec, issues with comments.

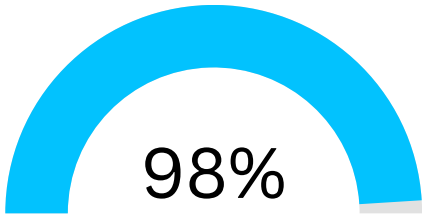
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



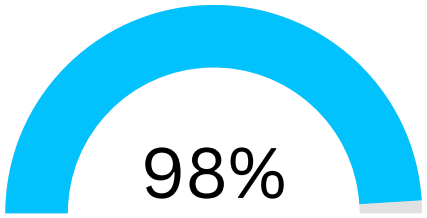
Security score

Security score is a numerical value generated based on the vulnerabilities in smart contracts. The score indicates the contract's security level and a higher score implies a lower risk of vulnerability.



Code quality

Code quality refers to adherence to standard practices, guidelines, and conventions when writing computer code. A high-quality codebase is easy to understand, maintain, and extend, while a low-quality codebase is hard to read and modify.



Documentation quality

Documentation quality refers to the accuracy, completeness, and clarity of the documentation accompanying the code. High-quality documentation helps auditors to understand business logic in code well, while low-quality documentation can lead to confusion and mistakes.

07-Findings

Finding: #1

Issue: USDC Blacklisted User Can Revert Block Verification Process.

Severity: High

Where: eth/contracts/rollup/RollupV6.sol

Impact: A blacklisted user set as **returnAddress** can block the verification process due to an unauthorized transfer

Description: The function **executeBurnToRouter** attempts to burn tokens by interacting with an external router and, if the router call fails, transfers the tokens to a designated **returnAddress**. However, there is a critical issue that arises if the **returnAddress** is blacklisted by the USDC token contract. Since USDC enforces blacklist functionality, any transfer to a blacklisted address will revert, causing the entire transaction to fail. This can lead to a Denial of Service (DoS) scenario, especially because this function is executed by a prover to verify a block in a rollup. If the transfer fails due to a blacklisted address, the block verification process would fail entirely.

```
function executeBurnToRouter(
    bytes32 nullifier,
    uint256 value,
    uint256 gasPerBurnCall
) internal returns (bool) {
    BurnToRouter memory b = burnsToRouter[nullifier];

    // This should never happen
    require(b.amount == value, "RollupV6: Invalid burn amount");

    IERC20(usdc).approve(b.router, value);
    if (!_routerCall(b.router, b.routerCalldata, gasPerBurnCall)) {
        // Call reverted.
        // Reset allowance to 0
        IERC20(usdc).approve(b.router, 0);

        // Return the funds to the return address
        IERC20(usdc).transfer(b.returnAddress, b.amount);

        // This is still a success, so we don't return false
    }

    return true;
}
```

If the **returnAddress** is blacklisted by USDC, the transfer will revert. Since there is no check in place to verify whether the **returnAddress** is blacklisted before attempting the transfer, this can cause an unexpected transaction failure. Therefore, as this function is primarily executed by a prover during rollup block verification, if it fails due to a blacklisted address, it would prevent the entire block from being verified. This could be exploited as a DoS attack vector against the rollup system.

```
function burnToRouter(
    bytes32 kind,
    bytes32 msgHash,
    bytes calldata proof,
    bytes32 nullifier,
    bytes32 value,
    bytes32 source,
    bytes32 sig,
    address router,
    // TODO: this could be big and use a lot of gas to set in storage, set a limit in guild
    bytes calldata routerCalldata,
    address returnAddress
) public override {
    require(kind == bytes32(uint256(1)), "Invalid kind");
    require(returnAddress != address(0), "Invalid return address");
    require(isRouterWhitelisted(router), "Router not whitelisted");

    bytes32 computedMsg = keccak256(
        abi.encode(router, routerCalldata, returnAddress)
    );
    // Clear the first 3 bits, BN256 can't fit the full 256 bits
    computedMsg &= bytes32(
        uint256(
            0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
        )
    );
    require(computedMsg == msgHash, "Invalid msg");

    burnVerifierV2.verify(
        proof,
        [kind, msgHash, nullifier, value, source, sig]
    );

    setBurnsKind(nullifier, kind);
    burnsToRouter[nullifier] = BurnToRouter(
        router,
        routerCalldata,
        uint256(value),
        returnAddress
    );
    emit BurnAdded(nullifier, uint256(value));
}
```

As it can be seen above, there is no check in charge of verifying if **returnAddress** is blacklisted.

Recommendations: Before storing a new **burnsToRouter** entry, the **returnAddress** address should be checked with the **requireNotUSDCBlacklisted** function.

Status: Resolved.

Finding: #2

Issue: Funds Get Stuck In Case Of Router Revert

Severity: Medium

Where: eth/contracts/rollup/RollupV6.sol

Impact: Funds will get stuck in contract if router's call fail.

Description: The function **executeBurnToRouter** is designed to execute a token burn by interacting with an external router, based on data stored in the **burnsToRouter** mapping. However, there is a potential issue that arises if the external call to the router fails. Specifically, when the router call fails, the function resets the token allowance to zero but does not transfer the tokens to any account. This leaves the tokens locked in the contract, which can lead to a loss of funds.

```
function executeBurnToRouter(
    bytes32 nullifier,
    uint256 value,
    uint256 gasPerBurnCall
) internal returns (bool) {
    BurnToRouter memory b = burnsToRouter[nullifier];

    // This should never happen
    require(b.amount == value, "RollupV6: Invalid burn amount");

    IERC20(usdc).approve(b.router, value);
    if (!_routerCall(b.router, b.routerCalldata, gasPerBurnCall)) {
        // Reset allowance to 0
        IERC20(usdc).approve(b.router, 0);
        return false;
    }

     return true;
}
```

As it can be seen above, if the external call executed in **_routerCall** fails, the function resets the token allowance but does not handle the remaining tokens. This means that tokens remain in the contract without being transferred or accessible by any user. Over time, these locked tokens could accumulate and become unrecoverable if their respective nullifiers are not reused.

Recommendations: Implement logic to transfer the remaining tokens back to the sender or another designated address if the router call fails. This ensures that no tokens are left locked in the contract.

Status: Resolved.

Finding: #3

Issue: Attacker Can Grief Another User's Router Burn

Severity: Low

Where: RollupV5.sol#L438-L45

Impact: An attacker can manipulate the **gasPerBurnCall** parameter in the **substituteBurn** function to interfere with another user's router burn operation. By passing arbitrary or malicious values for **gasPerBurnCall**, the attacker can disrupt the execution of the router call, potentially causing unexpected behavior or denial of service for the victim.

Description: The **substituteBurn** function allows users to specify the **gasPerBurnCall** parameter, which is subsequently used in the router burn operation:

```
function substituteBurn(
    bytes32 nullifier,
    uint256 amount,
    uint256 gasPerBurnCall
) public {
    ...
    (bool found, bool success) = executeBurn(
        nullifier,
        amount,
        gasPerBurnCall,
        true
    );
```

The **gasPerBurnCall** value directly affects the behavior of the **_routerCall** function:

```
function executeBurnToRouter(
    bytes32 nullifier,
    uint256 value,
    uint256 gasPerBurnCall
) internal returns (bool) {
    ...
    if (!_routerCall(b.router, b.routerCalldata, gasPerBurnCall)) {
```

Since any user can call **substituteBurn** and specify the **gasPerBurnCall**, an attacker can:

- Set a very low gas value, causing the router call to fail.
- Set an excessively high or invalid value to disrupt normal operations or waste gas.

This capability allows an attacker to interfere with other users' router burn processes, reducing the reliability of the protocol.

Recommendations:

- **Set a Default Gas Value:** Replace the user-specified **gasPerBurnCall** parameter with a predefined default value (e.g., 500,000 gas as used in **verifyBlock**):

```
uint256 constant DEFAULT_GAS_PER_BURN_CALL = 500_000;
```

- **Remove gasPerBurnCall Input from substituteBurn:** Hard-code the **DEFAULT_GAS_PER_BURN_CALL** value into the **executeBurnToRouter** call, ensuring consistent and predictable behavior:

```
(bool found, bool success) = executeBurn(
    nullifier,
    amount,
    DEFAULT_GAS_PER_BURN_CALL,
    true
);
```

Status: Resolved.

Finding: #4

Issue: Router Reentrancy Allows Bypassing Single-Time Substitutions

Severity: Low

Where: RollupV6.sol#L311-L337
RollupV6.sol#L231-L249

Impact: A malicious router can exploit a reentrancy vulnerability to repeatedly call the **substituteBurn** function, bypassing the one-time substitution restriction. This behavior could lead to unintended consequences, depending on the reasoning behind the one-time substitution limit. Although the severity is limited to the substitution logic, it still undermines the integrity of the burn process.

Description: The **substituteBurn** function is designed to allow a user to replace a burn recipient while ensuring it can only happen once for each burn. This is enforced by the **substitutedBurns** mapping, which tracks whether a burn has been substituted:

```
require(!substitutedBurns[nullifier], "RollupV6: Burn already substituted");
```

However, during the substitution process, an external call is made to the router via the **executeBurnToRouter** function:

```
if (!_routerCall(b.router, b.routerCalldata, gasPerBurnCall)) {
```

This external call introduces a reentrancy vulnerability, as the **substitutedBurns** mapping is only updated after the burn execution:

```
substitutedBurns[nullifier] = true;
```

A malicious router can exploit this gap to re-enter the **substituteBurn** function before the mapping is updated, allowing it to substitute the burn multiple times.

Recommendations:

- **Add a Reentrancy Guard:** Use a reentrancy guard to ensure that the **substituteBurn** function cannot be re-entered during execution. For example, integrate OpenZeppelin's **ReentrancyGuard**.
- **Follow Checks-Effects-Interactions (CEI) Pattern:** Update the **substitutedBurns** mapping before executing the external call to the router. This prevents reentrancy by ensuring the one-time substitution restriction is applied early in the process.

Status: Resolved.

Finding: #5

Issue: Attacker Can Grief **substituteBurn** Caller Leading to Locking Funds

Severity: Low

Where: RollupV6.sol#L311-L337

Impact: An attacker can exploit the timing of the **substituteBurn** process to lock a user's funds. By submitting a UTXO with the same nullifier before the user's substitution is included in a block, the attacker consumes the nullifier, making the user's transaction invalid and effectively preventing them from accessing their funds.

Description: The **substituteBurn** function allows users to substitute an existing burn by creating a new burn tied to the same nullifier:

```
burns[nullifier] = Burn({to: msg.sender, amount: amount});
```

After substitution, the user must include the burn's UTXO in a block through the **verifyBlock** function. However, an attacker monitoring the mempool can submit a UTXO with the same nullifier before the user's transaction is included in a block. This causes the user's transaction to fail since the nullifier is already consumed.
The attacker gains control of the funds, and the user's substitution becomes invalid, effectively locking their funds. Since finding a new valid nullifier for the same UTXO is not feasible, the user is permanently affected.

Recommendations: Modify the contract logic to mark burns as "finalized" or "executed" once they are included in a block via **verifyBlock**. This would prevent a finalized burn from being used in a **substituteBurn** operation.

Status: Resolved.

Finding: #6

Issue: Remaining Unused Approval Could Lead To Stolen Tokens

Severity: Low

Where: eth/contracts/rollup/RollupV6.sol

Impact: Unused approval by the router could lead to unexpected transfers by third parties.

Description: The function **executeBurnToRouter** interacts with an external router by approving a certain amount of tokens for transfer. However, there is a **residual approval risk** that arises when the router does not use the full approved amount of tokens. If the router call succeeds but does not consume all the approved tokens, the contract does not reset the approval back to zero. This leaves a residual token allowance, which can be exploited by third parties to drain the remaining approved tokens for their own benefit.

```
function executeBurnToRouter(
    bytes32 nullifier,
    uint256 value,
    uint256 gasPerBurnCall
) internal returns (bool) {
    BurnToRouter memory b = burnsToRouter[nullifier];

    // This should never happen
    require(b.amount == value, "RollupV6: Invalid burn amount");

    IERC20(usdc).approve(b.router, value);
    if (!_routerCall(b.router, b.routerCalldata, gasPerBurnCall)) {
        // Call reverted.
        // Reset allowance to 0
        IERC20(usdc).approve(b.router, 0);

        // Return the funds to the return address
        IERC20(usdc).transfer(b.returnAddress, b.amount);

        // This is still a success, so we don't return false
    }

    return true;
}
```

Therefore, after a successful router call, there is no guarantee that the full approved amount of tokens will be used by the router. If any portion of the approved tokens remains unused, it creates a residual approval that could be used by external actors.

Recommendations: After a successful call to the router, reset the token approval back to zero to prevent any residual approvals from being exploited. Also, another approach could be to transfer the remaining tokens to **returnAddress**.

Status: Resolved.

Finding: #7

Issue: Missing **BurnAdded** event in **burn** function

Severity: Low

Where: RollupV6.sol#L118-L139

Impact:

- Reduced ability to track burn operations through events
- Inconsistent event emission pattern across similar functions (**burnToAddress** and **burnToRouter** emit the event while **burn** doesn't)
- Potential issues with frontend applications that rely on events to track burn operations
- Harder to audit and track historical burn operations through event logs

Description: The **burn()** function in **RollupV6.sol** fails to emit the **BurnAdded** event after recording a new burn operation. This is inconsistent with other burn-related functions in the contract (**burnToAddress** and **burnToRouter**) which properly emit this event. The BurnAdded event is important for tracking burn operations and is used by the application to track which transaction the burn was added in, especially in cases where the transaction hash might be lost.

Recommendations: Add the **BurnAdded** event emission at the end of the **burn()** function:

```
function burn(
    address to,
    bytes calldata proof,
    bytes32 nullifer,
    bytes32 value,
    bytes32 source,
    bytes32 sig
) public override {
    .......
    .......
    emit BurnAdded(nullifer, uint256(value));  // Add this line
}
```

Status: Resolved.

Finding: #8

Issue: Lack Of NATSPEC

Severity: Quality Assurance

Where: All in-scope contracts

Impact: Improve readability and documentation.

Description: NATSPEC documentation to all public methods and variables is essential for better understanding of the code by developers and auditors and is strongly recommended.

Recommendations: Add NATSPEC comments to each function and variable to improve contracts' readability.

Status: Acknowledged.

Finding: #9

Issue: Use Custom Errors Instead Of Revert Strings

Severity: Quality Assurance

Where: All contracts in-scope

Impact: Gas optimization and code readability.

Description: Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met).

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

Starting from [Solidity v0.8.4], there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., **revert("Insufficient funds.");**), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the error statement, which can be used inside and outside of contracts (including interfaces and libraries).

Recommendations: Implement custom errors instead of revert strings.

Status: Acknowledged.

Finding: #10

Issue: Gas Optimization

Severity: Quality Assurance

Where:

- Use short circuit on RollupV6.sol#L262-L267:

```
success = found && executeBurnToAddress(mb, value); // Short circuit when found is false
```

- Early **return false** on **executeBurnToRouter**, if **b.amount == 0**, on RollupV6.sol#L239

```
// This should never happen
require(b.amount == value, "RollupV6: Invalid burn amount");
if (b.amount == 0) {
    return false;
}
```

- Avoid **require** on **USDC.transfer** and **USDC.transfeFrom**, given that USDC will ALWAYS return true after a transfer you can avoid the pattern:

```
require(
            IERC20(usdc).transferFrom(FROM, TO, AMOUNT),
            "RollupV6: Transfer failed"
        );
```

and just use

```
IERC20(usdc).transferFrom(FROM, TO, AMOUNT):
IERC20(usdc).transfer(TO, AMOUNT):
```

Status: Resolved.

08 - Disclaimer

The smart contracts provided to AuditOne have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions). The ethical nature of the project is not guaranteed by a technical audit of the smart contract. Any owner-controlled functions should be carried out by the responsible owner. Before participating in the project, all investors/users are recommended to conduct due research.

The focus of our assessment was limited to the code parts associated with the items defined in the scope. We draw attention to the fact that due to inherent limitations in any software development process and product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which cannot be free from any errors or failures. These preconditions can impact the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure, which adds further inherent risks as we rely on correctly executing the included third-party technology stack itself. Report readers should also consider that over the life cycle of any software product, changes to the product itself or the environment in which it is operated can have an impact leading to operational behaviors other than initially determined in the business specification.

Contact

auditone.io

@auditone_team

hello@auditone.io

A trust layer of our
multi-stakeholder world.