Zellic

LightWallet

Smart Contract Security Assessment

August 21, 2023

Prepared for:

Shun Kakinoki

Light, Inc.

Prepared by:

Aaron Esau and Yuhang Wu

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 Light, Inc. on August 17th, 2023. During this engagement, Zellic reviewed LightWallet'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 all parent contracts properly overriden?
- Does the LightWallet miss integrations for any important account abstraction functions?

1.2 Non-goals and Limitations

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

- The mechanism for verifying signatures in ModuleAuth
- EntryPoint nonce verification, or any other security features provided by it
- Other parent contracts' security, aside from those relevant to the integration

1.3 Results

During our assessment on the scoped LightWallet contracts, we discovered one finding, which was informational in nature.

Breakdown of Finding Impacts

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

2 Introduction

2.1 About LightWallet

LightWallet is a multichain ERC-4337 implementation.

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 the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

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:

LightWallet Contracts

Repository	https://github.com/LightDotSo/LightDotSo/
Version	LightDotSo: `ad1cbb387df9cc02344fdde4489fb58912e9dab9`
Programs	LightWalletFactory LightWallet
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 12 person-hours. The assessment was conducted over the course of one calendar day.

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:

Aaron Esau, Engineer
aaron@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 17, 2023	Start of primary review period
August 17, 2023	End of primary review period

3 Detailed Findings

3.1 Unnecessary casting of `salt` parameter

- **Target**: LightWalletFactory
- **Category**: Optimizations
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The type of the parameter `salt` in the `createAccount` and `getAddress` functions is `uint256`, but the functions both cast it to `bytes32` in all uses of the parameter.

Recommendations

The `salt` parameter's type can be directly set to `bytes32`, eliminating the need for type conversion within the functions:

```
function createAccount(bytes32 hash, uint256 salt) public returns (
    LightWallet ret) {
function createAccount(bytes32 hash, bytes32 salt) public returns (
    LightWallet ret) {
    address addr = getAddress(hash, salt);
    // [...]
    ret = LightWallet(
        payable(
            new ERC1967Proxy{salt : bytes32(salt)}(
            new ERC1967Proxy{salt : salt}(
              address(accountImplementation),
              abi.encodeCall(LightWallet.initialize, (hash))
            )
        )
    );
}

// [...]

function getAddress(bytes32 hash, uint256 salt) public view returns (
```

```
    address) {
function getAddress(bytes32 hash, bytes32 salt) public view returns (
    address) {
    // Computes the address with the given `salt`and the contract address
    `accountImplementation`, and with `initialize` method w/ `hash`
    return Create2.computeAddress(
        bytes32(salt),
        salt,
        keccak256(
            abi.encodePacked(
                type(ERC1967Proxy).creationCode,
                abi.encode(address(accountImplementation),
    abi.encodeCall(LightWallet.initialize, (hash)))
            )
        )
    );
}
```

Remediation

This issue has been acknowledged by Light, Inc., and a fix was implemented in commit 6a1a082e.

4 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1 Module: LightWalletFactory.sol

Function: `createAccount(byte[32] byte[32], uint256 uint256)`

This is a helper function used to get the address of a deployed LightWallet contract or deploy a new one.

Inputs

- `hash`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Specifies the EntryPoint address the LightWallet should use.
- `salt`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Specifies the salt to use when deploying the proxy contract for LightWallet.

Branches and code coverage (including function calls)

Intended branches

- Account already exists — return existing LightWallet.
  ☑ Test coverage
- Account does not exist — create new LightWallet.
  ☑ Test coverage

4.2 Module: LightWallet.sol

Function: `executeBatch(address[] dest, uint256[] value, byte[][] func)`

Executes a sequence of transactions (called directly by `entryPoint`).

Inputs

- `dest`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The array of the address of the target contract to call.
- `value`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The array of amount of Wei (ETH) to send along with the call.
- `func`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The array of calldata to send to the target contract.

Branches and code coverage (including function calls)

Intended branches

- Tests that the account can run `executeBatch` correctly.
  ☑ Test coverage
- Tests that the account can run `executeBatch` correctly with `value.length == 0`.
  ☑ Test coverage

Negative behavior

- Tests that the account reverts when running `executeBatch` from a non-`entryPoint`.
  ☑ Negative test
- Tests that the account reverts when `dest.length` is not equal with `func.length`.
  ☐ Negative test

Function call analysis

- `executeBatch` → `_call(address target, uint256 value, bytes memory data)` → `target.call{value: value}(data)`
  - **What is controllable?** `target`, `value`, and `data`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

If there is a reentry attempt, the function will revert because the `execute` method is called from a non-`entryPoint`.

Function: `execute(address dest, uint256 value, byte[] func)`

Executes a transaction (called directly by `entryPoint`).

Inputs

- `dest`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The address of the target contract to call.
- `value`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The amount of Wei (ETH) to send along with the call.
- `func`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The calldata to send to the target contract.

Branches and code coverage (including function calls)

Intended branches

- Tests that the account can run `execute` correctly.
  ☑ Test coverage

Negative behavior

- Tests that the account reverts when running `execute` from a non-`entryPoint`.
  ☑ Negative test

Function call analysis

- `execute` → `_call(address target, uint256 value, bytes memory data)` → `target.call{value: value}(data)`
  - **What is controllable?** `target`, `value`, and `data`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If there is a reentry attempt, the function will revert because the `execute` method is called from a non-`entryPoint`.

5 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped LightWallet contracts, we discovered one finding, which was informational in nature. Light, Inc. 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.