SMART CONTRACT AUDIT REPORT

for

LlamaPay

Prepared By: Xiaomi Huang

PeckShield

June 21, 2022

Document Properties

Client	LlamaPay
Title	Smart Contract Audit Report
Target	LlamaPay
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 21, 2022	Xuxian Jiang	Final Release
1.0-rc	June 16, 2022	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the LlamaPay protocol, we outline in this report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About LlamaPay

LlamaPay is a multi-chain protocol that allows you to automate transactions and stream them by the second. The recipients can withdraw these funds at any time. This eliminates the need for manual transactions. When used by employers to pay employees, employers will be able to deposit funds and create streams for employees. The employee starts getting paid the second that stream is created and can withdraw at any time.

Table 1.1: Basic Information of LlamaPay

Item	Description
Name	LlamaPay
Website	https://llamapay.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 21, 2022

In the following, we show the Git repositories of the reviewed files and the commit hash value used in this audit.

- https://github.com/LlamaPay/llamapay.git (527ece9)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/LlamaPay/llamapay.git (TBD)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [4], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `LlamaPay` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	2	🟦 🟦
Informational	0	
Total	2	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 low-severity vulnerabilities.

Table 2.1: Key LlamaPay Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested immutable Usages For Gas Efficiency	Coding Practices	Confirmed
PVE-002	Low	Improved Validation Of Function Arguments in _createStream()	Coding Practices	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Suggested immutable Usages For Gas Efficiency

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LlamaPay
- Category: Coding Practices [3]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, Solidity introduces the feature of declaring a state as immutable. An immutable state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as immutable is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an immutable state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of immutable states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show one key state variable token defined in LlamaPay contract. If there is no need to dynamically update it after the construction, it can be declared as either constant or immutable for gas efficiency. In particular, the state variable token can be defined as immutable as it will not be changed after its initialization in constructor().

```
contract LlamaPay is BoringBatchable {
    using SafeERC20 for IERC20;

    struct Payer {
        uint40 lastPayerUpdate;
```

```
            uint216 totalPaidPerSec; // uint216 is enough to hold 1M streams of 3e51
                tokens/yr, which is enough
        }

        mapping (bytes32 => uint) public streamToStart;
        mapping (address => Payer) public payers;
        mapping (address => uint) public balances; // could be packed together with
            lastPayerUpdate but gains are not high
        IERC20 public token;
        uint public DECIMALS_DIVISOR;
    ...
}
```

Listing 3.1: LlamaPay.sol

Note another state variable DECIMALS_DIVISOR shares the same issue and can be improved by defining it as immutable.

Recommendation Revisit the state variable definition and make extensive use of constant/ immutable states.

Status The issue has been confirmed by the team. And the team clarifies that: if these are immutable then etherscan doesn't automatically recognize new deploys (so we would have to manually verify etherscan code for each new contract that anyone deploys through factory).

3.2 Improved Validation Of Function Arguments in _createStream()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LlamaPay
- Category: Coding Practices [3]
- CWE subcategory: CWE-628 [2]

Description

The LlamaPay contract provides an internal _createStream() routine which will be invoked to create a new stream. To elaborate, we show below the related code snippet of the _createStream() routine. As the name indicates, it's designed for the sender (msg.sender) to create a new stream, which streams the transaction (amountPerSec) by the second to the recipient (to). So the recipient can then withdraw the funds at any time. While examining the validation of the input parameters to create a valid stream, we notice it doesn't validate the recipient address (parameter to). As a result, if the recipient address is address(0), the related funds may be locked in the contract, because transfering the ERC20 token to address(0) may revert. Based on this, we may need to validate the parameter to is not equal to address(0) at the beginning of the function.

```
    function _createStream(address to, uint216 amountPerSec) internal returns (bytes32
        streamId){
        streamId = getStreamId(msg.sender, to, amountPerSec);
        require(amountPerSec > 0, "amountPerSec can't be 0");
        require(streamToStart[streamId] == 0, "stream already exists");
        streamToStart[streamId] = block.timestamp;

        Payer storage payer = payers[msg.sender];
        uint totalPaid;
        uint delta = block.timestamp - payer.lastPayerUpdate;
        unchecked {
            totalPaid = delta * uint(payer.totalPaidPerSec);
        }
        balances[msg.sender] -= totalPaid; // implicit check that balance >= totalPaid,
            can't create a new stream unless there's no debt

        payer.lastPayerUpdate = uint40(block.timestamp);
        payer.totalPaidPerSec += amountPerSec;
    }
```

Listing 3.2: _createStream()

Recommendation Validate the parameter to is not equal to address(0) at the beginning of the _createStream() routine.

Status This issue has been confirmed by the team. And the team decides to leave it as it is, considering it's not a security issue but adding a check would increase gas costs for everyone.

4 Conclusion

In this audit, we have analyzed the design and implementation of the LlamaPay protocol, which is a multi-chain protocol that allows you to automate transactions and stream them by the second. And the recipients can withdraw these funds at any time. Which eliminates the need for manual transactions. The current code base is well organized and those identified issues are promptly confirmed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[3] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[4] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[5] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[6] PeckShield. PeckShield Inc. https://www.peckshield.com.