Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Lthememecoin

Veridise Inc.

July 13, 2023

| Prepared For:

Lthememecoin

| Prepared By:

Jacob Van Geffen

| Contact Us: `contact@veridise.com`

| Version History:

July 13th, 2023 Final
July 12th, 2023 Second Draft
July 8th, 2023 Initial Draft

Contents

1 Executive Summary

On June 30, 2023, Lthememecoin developers engaged Veridise to review the security of their L Token.The review covered the security and functional correctness of their ERC20-like token defined in `L.sol`. Veridise conducted the assessment over 1 person-day, with 1 engineers reviewing code over 1 day. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Lthememecoin developers provided the source code of the L token contracts for review. To facilitate the Veridise auditors' understanding of the code, the Lthememecoin developers interactively answered questions about the source contracts and intended functionality. The source code also contained some documentation in the form of documentation comments on functions and storage variables.

Summary of issues detected. The audit uncovered 3 issues, 0 of which are assessed to be of high or critical severity by the Veridise auditors. The Veridise auditors identified a medium-severity issue pertaining to front-running transactions that alter approved allowance amounts as well as a number of minor issues.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Lthememecoin	N/A	Solidity	N/A

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
June 30, 2023	Manual & Tools	1	1 person-day

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	2	2
Low-Severity Issues	0	0
Warning-Severity Issues	1	1
Informational-Severity Issues	0	0
TOTAL	3	3

Table 2.4: Category Breakdown.

Name	Number
Frontrunning	1
Logic Error	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Lthememecoin's smart contracts. In our audit, we sought to answer the following questions:

- Can users spend more than their allotted allowance for any other user?
- Can users manipulate transactions in order to increase their balance above what should be achieved through intended behavior?
- Does the anti-snipping mechanism work as intended?
- Does the `permit` transaction correctly recover the intended address?
- Does the transfer whitelist work as intended?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following technique:

- *Fuzzing/Property-based Testing.* We leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the `L.sol` and `PoolAddress.sol` files of the source code provided by the Lthememecoin developers, which contains the smart contract implementation of the Lthememecoin.

Methodology. Veridise auditors reviewed the reports of previous audits for Lthememecoin, inspected the provided tests, and read the Lthememecoin documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Lthememecoin developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

Table 3.4: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-L-VUL-001	approve transaction can be front-run	Medium	Acknowledged
V-L-VUL-002	Users can exceed anti-snipping limit . . .	Medium	Fixed
V-L-VUL-003	Storage variable “owner” should have a . . .	Warning	Acknowledged

3.4 Detailed Description of Issues

3.4.1 V-L-VUL-001: approve transaction can be front-run

Severity	Medium	**Commit**	N/A
Type	Frontrunning	**Status**	Acknowledged
File(s)		`L.sol`	
Location(s)		approve()	

As detailed by OpenZeppelin's `IERC20.sol` (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol), calls to `approve` may be front-run by the spender in order to use the balance that has already been approved before having more spending approved. In the current implementation, there is no guard against this.

```
function approve(address spender, uint256 amount) external returns (bool) {
    allowance[msg.sender][spender] = amount;

    emit Approval(msg.sender, spender, amount);

    return true;
}
```

Furthermore, there is currently no way to increase the approved allowance amount without directly setting a new value.

Impact If a spender is originally approved to spend `N` tokens on the approver's behalf, and a call to `approve(spender, M)` is made, then the spender can use this exploit to spend `N+M` tokens on the user's behalf. This enables malicious users to spend more than their intended allowance.

Recommendation There are two possible fixes to this problem:

1. Include two new external functions — `increaseAllowance` and `decreaseAllowance` — so that token holders may update allowance without the possibility of a front-run attack.
2. Require that each call to `approve` either sets `allowance[msg.sender][spender]` to 0, or that the previous value of `allowance[msg.sender][spender` was 0. This forces users to reset the allowance to 0 before changing the allowance amount, and thus preventing the front-run attack.

Since the first solution is more standard and straightforward (and has also been adopted by OpenZeppelin's `ERC20.sol` implementation), we recommend solution (1) over solution (2).

Developer Response The developers have confirmed the issue, but since ownership had already been renounced, it cannot be implemented.

3.4.2 V-L-VUL-002: Users can exceed anti-snipping limit through use of multiple accounts

Severity	Medium	Commit	N/A
Type	Logic Error	Status	Acknowledged
File(s)		`L.sol`	
Location(s)	_beforeTokenTransfer(), setAntiSnipping()		

In order to prevent snipping, `setAntiSnipping(...)` defines a maximum value that any user can buy from a particular liquidity pool. This value is defined per-pool in the `antiSnipping` map:

```
function setAntiSnipping(address factory, address tokenA, address tokenB, uint24 fee,
     uint256 value) external onlyOwner returns (address pool) {
    PoolAddress.PoolKey memory poolKey = PoolAddress.getPoolKey(tokenA, tokenB, fee);

    pool = PoolAddress.computeAddress(factory, poolKey);

    antiSnipping[pool] = value;

    emit AntiSnippingSet(pool, value);
}
```

The mechanism behind enforcing anti-snipping is implemented in `_beforeTokenTransfer()`, and checks to see whether or not the new balance of the resulting transfer exceeds the bound defined by the `antiSnipping` map.

```
function _beforeTokenTransfer(address from, address to, uint256 amount) view internal
     {
    if (!transferable) {
        require(whitelist[from] || whitelist[to], "INVALID_WHITELIST");
    }

    if (antiSnipping[from] > 0) {
        require(balanceOf[to] + amount <= antiSnipping[from], "BALANCE_LIMIT");
    }
}
```

However, users can easily circumvent this bound in order to buy as many tokens from the pool as they wish using the following scheme:

1. Control two accounts `A` and `B`.
2. Buy `antiSnipping[pool]` tokens using account `A` from `pool`.
3. Transfer all balance from `A` to `B`.
4. Repeat from step (2) as desired.

Additionally, users that may have a balance that is independently larger than `antiSnipping[pool]` are prevented from buying any tokens from the pool, regardless of whether or not they have previously bought pool tokens.

Impact Since the cost of circumventing the anti-snipping restrictions is relatively low — as it only requires users generate one additional account and pay the gas fees of transferring between those accounts — malicious users can perform an unlimited amount of snipping.

Additionally, good-acting users wishing to buy from the pool may be incorrectly prevented if their balance is too high.

Recommendation In addition to the `antiSnipping` map, keep track of an additional mapping `balanceFromPool`, which tracks how much each account has purchased from each pool

```
mapping (address => mapping(address => uint256)) public balanceFromPool;
```

The requirement in `_beforeTokenTransfer` could then be updated based on this new mapping:

```
function _beforeTokenTransfer(address from, address to, uint256 amount) view internal
     {
    if (!transferable) {
        require(whitelist[from] || whitelist[to], "INVALID_WHITELIST");
    }

    if (antiSnipping[from] > 0) {
        require(balanceFromPool[to][from] + amount <= antiSnipping[from], "
    BALANCE_LIMIT");
    }
}
```

Developers must also update this mapping whenever a `transfer` or `transferFrom` involving a pool succeeds.

The result of this fix will be that users wishing to buy more than the anti-snipping limit will have to generate many more accounts (specifically, one account per `antiSnipping[pool]` they wish to buy). This additional overhead will de-incentivize users from exceeding this limit. Also, since a user's balance is kept separate from the balance gained from the pool, good-acting users can buy tokens up to the limit from a `pool` no matter their previous balance of `L`.

Developer Response The developers have confirmed the issue. The anti-snipping limit has been lifted, and thus this is no longer an issue.

3.4.3 V-L-VUL-003: Storage variable "owner" should have a different name

Severity	Warning	**Commit**	N/A
Type	Maintainability	**Status**	Acknowledged
File(s)		`L.sol`	
Location(s)		permit()	

The storage variable `owner` refers to the owner of the contract. However, the `permit()` function also takes `owner` as a parameter, referring to the user who's tokens should be permitted for spending by the `spender` account.

```
1 function permit(address owner, address spender, uint256 value, uint256 deadline,
      uint8 v, bytes32 r, bytes32 s) external
```

The names of one of these variables should be changed to avoid confusion and the possibility of future bugs.

Impact Future refactors may cause bugs if not all occurrences of the `owner` variable are changed. For example, if the `owner` parameter in the `permit()` function is refactored to a different name but some parts of permit still reference `owner`, `L.sol` will successfully compile despite the incomplete refactor.

Recommendation Change the name of storage variable `owner` to `contractOwner` so that future confusion can be avoided. If developers still wish for the view function to be named `owner()`, implement a new view function `owner()` that returns the value of `contractOwner`.

Developer Response The developers have acknowledged the issue.

4 Fuzz Testing

4.1 Methodology

Our goal was to fuzz test the L token to assess its functional correctness i.e, whether the implementation deviates from the intended behavior. We used OrCa, our in-house fuzzer, to verify invariants – logical formulas that should hold after every transaction. For each invariant, we wrote a harness which executed a random sequence of relevant external calls and then asserted the invariant should hold after the calls. We prioritized invariants which had a higher security impact. For all invariants, we ran OrCa for 10 minutes with 10 simulated users.

4.2 Properties Fuzzed

Table 4.1 describes the invariants we fuzz-tested. The first column states which contract the invariant is associated with. The second describes the invariant informally in English, and the last column notes whether we found a bug when fuzzing the invariant (✗ indicates no bug was found and X means fuzzing this invariant revealed a bug). We ran OrCa for 10 minutes when fuzz-testing each invariant.

Table 4.1: Invariants Fuzzed.

Contract	Invariant	Bug
L	Transfer reverts if the user attempts to send more funds than they have.	✗
L	Funds transfer to sender when the sender != receiver.	✗
L	transfer and transferFrom should not modify irrelevant state.	✗
L	approve appropriately updates state.	✗
L	burn will revert if user does not have enough funds.	✗
L	The unchecked block of _mint would never revert if checked.	✗
L	The unchecked block of burn would never revert if checked.	✗
L	The unchecked block of transfer would never revert if checked.	✗
L	The unchecked block of transferFrom would never revert if checked.	✗

As shown in the table above, no violations of the invariants above were found through fuzz testing. Additional manual checking of these invariants similarly found no violations.