Leverj Gluon Plasma

Audit Report

LEVERJ

BLOCKCHAIN LABS NZ

Table of contents

Background

This audit report was undertaken by **BlockchainLabs.nz** for the purpose of providing feedback to **Leverj**. It has subsequently been shared publicly without any express or implied warranty.

Solidity contracts were provided by the Leverj development team at this commit [leverj-gluon-plasma @ a44c9cd476539ed9d68df369cfef5b410ed1afa3] and the most recent commit we have audited is this [41a78e6ea77c17cf06e9180e52e1848dfc168c7e]- we would encourage all community members and token holders to make their own assessment of the contracts.

The audit was performed in **two phases**:

1. System analysis
2. Security audit *- this document*

Document structure

The report will include the following sections:

- Security audit
- Observations
- Conclusion

Focus areas

Correctness	No correctness defects uncovered during static analysis?
	No implemented contract violations uncovered during execution?
	No other generic incorrect behaviour detected during execution?
	Adherence to adopted standards such as ERC20?
Testability	Test coverage across all functions and events?
	Test cases for both expected behaviour and failure modes?
	Settings for easy testing of a range of parameters?
	No reliance on nested callback functions or console logs?
	Avoidance of test scenarios calling other test scenarios?
Security	No presence of known security weaknesses?
	No funds at risk of malicious attempts to withdraw/transfer?
	No funds at risk of control fraud?
	Prevention of Integer Overflow or Underflow?
Best Practice	Explicit labeling for the visibility of functions and state variables?
	Proper management of gas limits and nested execution?
	The latest version of the Solidity compiler?

Scope

All smart contracts, test files, migration and deployment scripts from this Github repo:
https://github.com/leverj/gluon-plasma/tree/master/packages/on-chain

Configuration files, documents and other assets were **out of the scope** of this audit.

Custodian.sol

Depositing.sol (interface)

Ledger.sol

Withdrawing.sol (interface)

ApiKeyRegistry.sol

Registry.sol (interface)

Fee.sol

Stake.sol

HasOwners.sol

Stoppable.sol

Switchable.sol

Validating.sol

Versioned.sol

Issues

Severity description

Minor	A defect that does not have a material impact on the contract execution and is likely to be subjective.
Moderate	A defect that could impact the desired outcome of the contract execution in a specific scenario.
Major	A defect that impacts the desired outcome of the contract execution or introduces a weakness that may be exploited.
Critical	A defect that presents a significant security vulnerability or failure of the contract across a range of scenarios.

Minor

- canExit() can be used in exit() 🏷️ *Best Practice*

```
235  function canExit(bytes32 entryHash) public view returns (bool) {
236    return
237      exitClaims[entryHash].confirmationThreshold != 0 &&  // exists
238      currentGblockNumber >= exitClaims[entryHash].confirmationThreshold;
239  }
```

'canExit()' at line 235 of 'Custodian.sol' verifies if the user can exit. It is a public function and it can be used in 'exit()' to replace line 207-208.

✔️Fix at 41a78e6e

- canSubmit() is always true 🏷️ *Correctness*

```
282  function canSubmit() public view returns (bool) { return block.number >=
submissionBlock; }
```

The 'canSubmit()' always returns 'true' because the variable 'submissionBlock' is set to equal to 'block.number' at contract deployment. From here forwards, 'block.number' will always be equal to or greater than 'submissionBlock', which makes the returned value 'true' for all time.

✔️Fix at 41a78e6e

- Gas optimisation of Deposit Commitment Record 🏷️ *Enhancement*

```
86  function deposit(address account, address asset, uint quantity) private whenOn {
87    uint nonce = ++nonceGenerator;
88    uint designatedGblock = currentGblockNumber + visibilityDelay;
89    DepositCommitmentRecord memory record = toDepositCommitmentRecord(account,
asset, quantity, nonce, designatedGblock);
90    deposits[record.hash] = true;
91    emit Deposited(address(this), account, asset, quantity, nonce,
designatedGblock);
92  }
```

Line 89 'DepositCommitmentRecord' is generated here, but only 'record.hash' is being used. To save some gas (for storing variable in memory), you could replace with below:

'hash = keccak256(abi.encodePacked(address(this), account, asset, quantity, nonce, designatedGblock));'

▢ Fix not required - works as designed

- submissionInterval is never used 🏷️ *Correctness*

'submissionInterval' is set at constructor but never used.

✔️ Fix at 41a78e6e

- double ternary operators are used 🏷️ *Best Practice*

```
uint withdrawLev = signedQuantity >= 0 ?
      0 :
      uint(signedQuantity * -1) >= stake.quantity ?
        stake.quantity :
        uint(signedQuantity * -1);
```

Ternary operators are use twice here, which makes the readability and maintainability harder.

There are many better approaches to improve it.

✔️ Fix at 41a78e6e

- Any one can withdraw() other people's asset 🏷️ *Enhancement*

```
function withdraw(address[] addresses, uint[] uints, bytes signature, bytes proof,
bytes32 root) external {
  Entry memory entry = extractEntry(addresses, uints);
  verifySignedBy(entry.hash, signature, operator);
  require(entry.entryType == EntryType.Withdrawal, "entry must be of type
Withdrawal");
  require(proveInConfirmedWithdrawals(proof, root, entry.hash), "invalid entry
proof");
  require(!withdrawn[entry.hash], "entry already withdrawn");
  withdrawn[entry.hash] = true;
  transfer(entry.quantity, entry.asset, entry.account);
  emit Withdrawn(entry.hash, entry.account, entry.asset, entry.quantity);
}
```

e.g. There are two users - A and B. User A deposits some asset in the ledger. Once user B get the proof/signature

from the operator, he can call 'withdraw()' function to transfer the asset back to user A.
In this case, user A doesn't actually lose anything, but it is kind of unexpected that the balance is not in his ledger account.
✔️Fix at [41a78e6e]()

- Variables could benefit from better naming and docstring 🏷️ *Best Practice*

Some params are sharing the name but not intrinsic meaning.
e.g. 'root' in 'claimExit()' means balances root, but in 'withdraw()' it means withdrawal root.
Recommend to have a better naming, or in order to keep variable's name short, add docstring in front of functions to clarify those variables(like what them mean and where they are from if necessary). That's a good practice for both community and your future maintainability.
▢ Fix not required - Future enhancement

Moderate

- None found

Major

- Don't trust outside contract 🏷️ *Security*

```
function depositToken(address token, uint quantity) external validToken(token) {
  require(Token(token).transferFrom(msg.sender, this, quantity), "failure to transfer quantity from token");
  deposit(msg.sender, token, quantity);
}
```

When a contract is calling functions in an outside contract, we need to verify if the action is really being done instead of trust the return value because the return value can be fake.
In a ERC20 compliant contract there is a 'transferFrom()'. We are expecting it increase Custodian's token balance while decrease executor's balance, and then return 'true' if transfer is success. However, if a dummy token contract designed on purpose with the 'transferFrom()' not really transferring any token but simply return 'true', Custodian cannot get transferred tokens in this dummy token but still increase executor’s token amount generating from nowhere in Custodian.
✔️Fix at [41a78e6e]()

Critical

- None found

Observations

It is documented that if the root hashes do not match the values committed to the plasma contract, there is a potential data unavailability issue and the participant can vote to halt. Currently there are no specified functions implementing the logic in the contracts. When the exception happens, there is one method less to protect the users by themselves. Considering the DApp is still under development, we are expecting more related code to be added in future releases to fulfill the objective.

When declaring the inheritance, it is very important to be aware that HasOwners contract needs to be always after Stoppable contract. Otherwise, the logic of onlyOwner modifier will be overwritten. At the moment of this audit, the code has been tested for success and failures of the onlyOwner modifier. In case future maintainers rewrite the code, they should ensure the tests for correct inheritance of onlyOwner are passing.

MerkleProof.sol is an imported external contract based on this repo. Leverj made some tweaks and added functions in the contract for convenience. However, these are only minor and do not affect the correctness of the merkle root functions.

Conclusion

We are satisfied that these Smart Contracts do not exhibit serious security vulnerabilities. Overall the code is well written and the developers have been responsive and active throughout the audit process. The contracts show care taken by the developers to follow best practices and a strong knowledge of Solidity. There is very high test coverage which should increase confidence in the security of these contracts, and their maintainability/development in the future.
