SHERLOCK

Security Review For Levr

Collaborative Audit Prepared For: **Levr**
Lead Security Expert(s): **nmirchev8**
Oblivionis
Date Audited: **November 4 - November 7, 2025**

Introduction

Levr turns any Clanker token into a community-owned project with built-in governance, staking rewards, and transparent treasury management.

For Token Holders:

- Earn rewards by staking, hard assets and project token
- Vote on proposals with time-weighted power
- Optional gasless transactions

For Project Creators:

- Launch or use an existing clanker token with instant DAO and staking capabilities
- Community ownership out of the box
- No token migration needed
- Fee splitter enables developers to send a portion of fees to stakers

For Developers:

- TypeScript SDK with React hooks
- Server-side APIs
- Full type safety

Scope

Repository: quantidexyz/levr-sc

Audited Commit: 25ca0362642e1dbb5b1c235a9f9bbe1c55b0611a

Final Commit: 0037b0673ba1b647d17555b57407fe5843d44a65

Files:

- script/DeployLevrFeeSplitter.s.sol
- script/DeployLevr.s.sol
- script/TransferFactoryOwnership.s.sol
- src/base/ERC2771ContextBase.sol
- src/interfaces/ILevrDeployer_v1.sol
- src/interfaces/ILevrFactory_v1.sol
- src/interfaces/ILevrFeeSplitterFactory_v1.sol
- src/interfaces/ILevrFeeSplitter_v1.sol
- src/interfaces/ILevrForwarder_v1.sol

- src/interfaces/ILevrGovernor_v1.sol
- src/interfaces/ILevrStakedToken_v1.sol
- src/interfaces/ILevrStaking_v1.sol
- src/interfaces/ILevrTreasury_v1.sol
- src/LevrDeployer_v1.sol
- src/LevrFactory_v1.sol
- src/LevrFeeSplitterFactory_v1.sol
- src/LevrFeeSplitter_v1.sol
- src/LevrForwarder_v1.sol
- src/LevrGovernor_v1.sol
- src/LevrStakedToken_v1.sol
- src/LevrStaking_v1.sol
- src/LevrTreasury_v1.sol
- src/libraries/RewardMath.sol

Final Commit Hash

0037b0673ba1b647d17555b57407fe5843d44a65

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
5	2	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Staking initialization front-run [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/23

Summary

An attacker can front-run the project owner during deployment and call `LevrStaking_v1.initialize(...)` first, supplying an arbitrary `factory_` and related params. This permanently locks the staking instance to attacker-controlled settings and bricks the project before `register()` finalizes.

Vulnerability Detail

Impact

DoS

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/c41ce83612f6dd6225cb88e9a1900b88fada2984/levr-sc/src/LevrFactory_v1.sol#L72-L73

Tool Used

Manual Review

Recommendation

Bind the legitimate factory at construction (immutable), remove the caller-supplied factory_ parameter, and require msg.sender == factory (or authorized deployer) for initialization.

Discussion

mguleryuz

pr => https://github.com/quantidexyz/levr-sc/pull/9

Issue H-2: Winner proposal can block the governance [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/25

Vulnerability Detail

- Malicious `token.balanceOf` hard-revert (definite blocker) Balance check is outside try/catch; any revert bubbles and reverts the whole execute, leaving the proposal Succeeded and preventing cycle advance.

```
uint256 treasuryBalance = IERC20(proposal.token).balanceOf(treasury);
```

- Gas bomb in token transfer/approve (conditional blocker) Calls happen inside try/catch via Treasury; normal reverts are caught and governance progresses. But if the callee consumes (nearly) all gas, or returns huge revert data causing OOG while copying revert payload in catch, the tx reverts, rolling back the "executed=true" state and blocking
- Revert data (return-calldata) bomb (conditional blocker) catch(Error(string)) and catch(bytes) copy revert data; extremely large revert payloads can OOG during copy, reverting the tx and blocking.

Impact

Governance DoS

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/c41ce83612f6dd6225cb88e9a1900b88fada2984/levr-sc/src/LevrGovernor_v1.sol#L175-L176

Tool Used

Manual Review

Recommendation

- Wrap balance check in a safe low-level `staticcall` with bounded return-data copy; on failure, mark proposal defeated instead of reverting.
- For execution, call Treasury methods via low-level call with gas cap and ignore revert data (no bytes binding), or have Treasury handle/catch token errors and return a bounded error code.

Discussion

mguleryuz

pr => https://github.com/quantidexyz/levr-sc/pull/8

Issue H-3: The pool's accounting allows a single user to claim all the rewards belonging to other users [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/26

Summary

The `claimRewards()` function uses a pool-based reward distribution model where users receive a proportional share of the current available pool. There is no mechanism to prevent users from calling this function multiple times, allowing them to drain the entire reward pool before other legitimate stakers can claim their fair share.

Vulnerability Detail

```
function claimRewards(address[] calldata tokens, address to) external nonReentrant {
        ...
        uint256 claimable = RewardMath.calculateProportionalClaim(
            userBalance,
            cachedTotalStaked,
            tokenState.availablePool
        );

        if (claimable > 0) {
            tokenState.availablePool -= claimable;  // @Audit Pool decreases
            IERC20(token).safeTransfer(to, claimable);
            emit RewardsClaimed(claimer, to, token, claimable);
        }
    }
}
```

In `LevrStaking_v1::claimRewards()`, there is no logic to prevent users from claiming multiple times within a single cycle, resulting in users receiving `share × (remaining pool / total share)` each time they claim.

Impact

Users can drain reward pool through repeated claims, stealing rewards from other stakers.

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/main/levr-sc/src/LevrStaking_v1.sol#L190

Proof of Concept

```
function test_EDGE_multipleClaimsGeometricDecrease() public {
    address[] memory tokens = new address[](1);
    tokens[0] = address(weth);

    address bob = address(0x2222);
    underlying.mint(bob, 1000 ether);
    weth.mint(bob, 1000 ether);

    // Alice stakes 500 tokens (50% of pool)
    vm.prank(alice);
    underlying.approve(address(staking), 500 ether);
    vm.prank(alice);
    staking.stake(500 ether);

    // Bob stakes 500 tokens (50% of pool)
    vm.prank(bob);
    underlying.approve(address(staking), 500 ether);
    vm.prank(bob);
    staking.stake(500 ether);

    // Accrue 1000 WETH rewards
    weth.transfer(address(staking), 1000 ether);
    staking.accrueRewards(address(weth));
    skip(7 days); // Make all rewards available (streamWindowSeconds = 7 days)

    console2.log("=== Testing Geometric Decrease (50% Stake) ===");
    console2.log("Initial pool: 1000 WETH");
    console2.log("Alice stake: 50%%, Bob stake: 50%%");
    console2.log("");

    uint256 totalAliceClaimed = 0;

    // Alice claims 10 times rapidly
    for (uint256 i = 1; i <= 10; i++) {
        uint256 balanceBefore = weth.balanceOf(alice);
        vm.prank(alice);
        staking.claimRewards(tokens, alice);
        uint256 claimed = weth.balanceOf(alice) - balanceBefore;
        totalAliceClaimed += claimed;

        console2.log("Alice claim %s: %s WETH (cumulative: %s)",
```

```
            i,
            claimed / 1e18,
            totalAliceClaimed / 1e18
        );
    }
    // Bob claims his share (should still be able to claim)
    uint256 bobBalanceBefore = weth.balanceOf(bob);
    vm.prank(bob);
    staking.claimRewards(tokens, bob);
    uint256 bobClaimed = weth.balanceOf(bob) - bobBalanceBefore;

    console2.log("");
    console2.log("=== Final Results ===");
    console2.log("Alice total: %s WETH (after 10 claims)", totalAliceClaimed );
    console2.log("Bob total: %s WETH (after 1 claim)", bobClaimed );
    console2.log("Pool dust: %s WETH", staking.claimableRewards(alice,
    ↪  address(weth)) );
}
```

The execution result is:

```
Ran 1 test for test/unit/LevrStakingV1.Accounting.t.sol:LevrStakingV1_Accounting
[PASS] test_EDGE_multipleClaimsGeometricDecrease() (gas: 606511)
Logs:
  === Testing Geometric Decrease (50% Stake) ===
  Initial pool: 1000 WETH
  Alice stake: 50%, Bob stake: 50%

  Alice claim 1: 500 WETH (cumulative: 500)
  Alice claim 2: 250 WETH (cumulative: 750)
  Alice claim 3: 125 WETH (cumulative: 875)
  Alice claim 4: 62 WETH (cumulative: 937)
  Alice claim 5: 31 WETH (cumulative: 968)
  Alice claim 6: 15 WETH (cumulative: 984)
  Alice claim 7: 7 WETH (cumulative: 992)
  Alice claim 8: 3 WETH (cumulative: 996)
  Alice claim 9: 1 WETH (cumulative: 998)
  Alice claim 10: 0 WETH (cumulative: 999)

  === Final Results ===
  Alice total: 999023437500000000000 WETH (after 10 claims)
  Bob total: 488281250000000000 WETH (after 1 claim)
  Pool dust: 244140625000000000 WETH

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.95ms (639.43µs CPU
↪  time)

Ran 1 test suite in 8.36ms (1.95ms CPU time): 1 tests passed, 0 failed, 0 skipped
↪  (1 total tests)
```

Tool Used

Manual Review

Recommendation

Consider introducing index-based accounting, maintaining a global cumulative index that records “the total rewards accumulated per unit of staked token.”

```
globalIndex = Σ(rewardRate × time / totalStaked)
userPending = userBalance × (globalIndex - userLastIndex)
```

Discussion

mguleryuz

pr => https://github.com/quantidexyz/levr-sc/pull/7

Issue H-4: `stake()` does not settle users' previous rewards, allowing attackers to drain the reward pool via flash loans [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/27

Summary

`LevrStaking_v1::stake()` function fails to call `_claimAllRewards()` for existing stakers before updating the total stake amount. As a result, when users stake additional tokens, previously accumulated rewards for existing balance are not settled. This omission allows new stakers to effectively dilute the reward distribution by entering the pool before pending rewards are properly accounted for.

Vulnerability Detail

`stake()` does not trigger `_claimAllRewards()` for existing stakers, allowing new stakers to instantly dilute the reward pool without settling existing claims first.

```
// stake() - No claim for existing stakers
function stake(uint256 amount) external nonReentrant {
    _settleAllPools();
    //@Audit missing _claimAllRewards(staker, staker) for existing balance

    _totalStaked += amount;
    ILevrStakedToken_v1(stakedToken).mint(staker, actualReceived);
}

// unstake() - Auto-claims everything
function unstake(uint256 amount, address to) external nonReentrant {
    _claimAllRewards(staker, to);
    // ... burn and transfer ...
}
```

`_claimAllRewards()` only accounts for the user's instantaneous balance. This means that if previous rewards are not settled, an attacker can, within a single transaction:

1. Take a flash loan and convert it into the underlying token
2. Deposit it into LevrStaking_v1, then immediately withdraw
3. Capture the majority of the pool's pending rewards before repaying the loan

Impact

A single attacker can directly drain the reward pool.

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/main/levr-sc/src/LevrStaking_v1.sol#L111-L151

Tool Used

Manual Review

Recommendation

When a user's staked balance changes (through staking or unstaking), all pending rewards must be settled, and the new rewards should start accumulating based on the updated balance.

Discussion

mguleryuz

pr => https://github.com/quantidexyz/levr-sc/pull/6

Issue H-5: Quorum calculation is based on instantaneous balances, allowing attackers to use flash loans to pass proposals that would otherwise fail to meet the quorum [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/29

Summary

The quorum mechanism counts immediate staked token balances via `balanceOf` rather than voting power, enabling flash‑loan amplification of participation during the voting window. An attacker can momentarily stake borrowed tokens to inflate `totalBalanceVoted` without increasing their actual voting power, causing proposals to appear to meet quorum illegitimately.

Vulnerability Detail

Within vote, voting power is measured using time‑weighted balance from `getVotingPower`, but quorum is measured by accumulating the caller's instantaneous staked token balance.

This discrepancy allows a malicious user to take a flash loan → deposit → vote → withdraw → repay the loan, inflating their instantaneous balance while keeping their voting power unchanged, leaving an artificially inflated participation count that persists for quorum evaluation. Since `meetsQuorum` compares `totalBalanceVoted` against a threshold derived from supply snapshots and basis points, the temporary balance manipulation can force quorum to be met even when genuine participation is low.

Impact

An attacker can cause a proposal that should have failed to bypass the quorum check. If executed in the last block of the voting window, this could potentially force through a malicious proposal that would otherwise go unnoticed.

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/main/levr-sc/src/LevrGovernor_v1.sol#L422

Tool Used

Manual Review

Recommendation

Quorum calculation should be based on voting power rather than instantaneous balance.

Discussion

mguleryuz

pr => https://github.com/quantidexyz/levr-sc/pull/10

Issue M-1: Trusted Clanker factory validation bypass when trusted list contains only non-deployed addresses [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/30

Vulnerability Detail

If owner sets _trustedClankerFactories to addresses with no code, `hasDeployedFactory` stays false and token validation is skipped. Any token admin can register an arbitrary token without verification:

```
require(_trustedClankerFactories.length > 0, "NO_TRUSTED_FACTORIES");
bool validFactory = false;
bool hasDeployedFactory = false;

for (uint256 i = 0; i < _trustedClankerFactories.length; i++) {
    address factory = _trustedClankerFactories[i];
    uint256 size;
    assembly { size := extcodesize(factory) }
    if (size == 0) continue;
    hasDeployedFactory = true;
    try IClanker(factory).tokenDeploymentInfo(clankerToken) returns (...) { ... }
}

if (hasDeployedFactory) {
    require(validFactory, "TOKEN_NOT_FROM_TRUSTED_FACTORY");
}
```

Impact

Validation bypass

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/c41ce83612f6dd6225cb88e9a1900b88fada2984/levr-sc/src/LevrFactory_v1.sol#L101-L114

Tool Used

Manual Review

Recommendation

Always require `validFactory == true` if any trusted factory addresses exist, regardless of code size; or validate that at least one trusted factory has code at configuration time and otherwise block registration until corrected.

Discussion

mguleryuz

addressed by => https://github.com/quantidexyz/levr-sc/pull/5

Issue M-2: Centralisation risk [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/32

Vulnerability Detail

Protocol admin can execute whatever proposal he wants by calling `factory::updateProjectConfig` right before cycle increase and setting voting period to 1 sec and `quorumBps` & `approvalBps` to 0

Impact

Centralisation risk

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/c41ce83612f6dd6225cb88e9a1900b88fada2984/levr-sc/src/LevrFactory_v1.sol#L221-L237

Tool Used

Manual Review

Recommendation

Enforce lower bounds and invariants in config Require voting/proposal windows sane mins (e.g., 1 day).

Issue L-1: Malicious executors can abuse EIP-150 to force a proposal to fail during execution [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/28

Summary

`LevrGovernor_v1::execute()` uses try-catch to handle proposal execution, but doesn't validate that sufficient gas was provided. An attacker can exploit EIP-150's 63/64 rule to cause `_executeProposal()` to fail due to out-of-gas while the try-catch succeeds, permanently marking the proposal as executed without actually transferring funds.

Vulnerability Detail

The `execute()` function uses try-catch to handle proposal execution, but doesn't validate that sufficient gas was provided. An attacker can exploit EIP-150's 63/64 rule to cause `_executeProposal()` to fail due to out-of-gas while the try-catch succeeds, permanently marking the proposal as executed without actually transferring funds.

According to EIP-150, when making an external call, the caller retains at least 1/64 of the gas. An attacker can calculate a precise gas value `G` such that:

- The pre-execution checks complete successfully
- The gas forwarded to `this._executeProposal()` (63G/64) is insufficient to complete treasury operations
- The remaining 1G/64 gas is enough for the `catch` block to capture the exception and start a new cycle

Impact

This issue will not be triggered at the moment because the gas consumed within the try-catch (1/63) is insufficient for the subsequent calculations. However, if a complex ERC20 token is used or if future updates introduce other proposal logic, it could potentially be triggered.

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/main/levr-sc/src/LevrGovernor_v1.sol#L199-L213

Tool Used

Manual Review

Recommendation

Ensure that no high-gas operations are introduced within the try-catch block in future updates.

Discussion

mguleryuz

pr => https://github.com/quantidexyz/levr-sc/pull/8

Issue L-2: Document a situation where winner proposal ratio matches another proposal ratio [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/31

Summary

If there are two proposals with the same approval ratio, the first one will be marked as winner

Vulnerability Detail

`_getWinner` in the governor contract iterates over all proposals and calculate their `approvalRatio`. It updates `bestApprovalRatio` only if the current is >. This means that if we have two proposals with ratio of 10_000, the first one will be the winner.

However, it is not clear that the following is a desired behavior.

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/c41ce83612f6dd6225cb88e9a1900b88fada2984/levr-sc/src/LevrGovernor_v1.sol#L478-L482

Tool Used

Manual Review

Recommendation

Document the behavior, or change it

Issue L-3: `LevrGovernor_v1::_state()` will treat all proposals that meet quorum and approval thresholds as succeeded [RESOLVED]

Source: https://github.com/sherlock-audit/2025-11-levr-nov-4th/issues/33

Summary

`LevrGovernor_v1::_state()` returns Succeeded for any proposal that meets quorum and approval thresholds, regardless of whether it's the cycle winner. This causes `getProposal` to treat all proposals that meet the quorum and approval thresholds as successful, even though only the winner can actually be executed.

Vulnerability Detail

`LevrGovernor_v1::_state` iterates over a proposal that is not the current cycle's winner but still meets the approval and quorum thresholds, then the entire workflow attempting to start a new cycle will be blocked.

```
function _state(uint256 proposalId) internal view returns (ProposalState) {
    ILevrGovernor_v1.Proposal storage proposal = _proposals[proposalId];

    if (proposal.id == 0) revert InvalidProposalType();
    if (proposal.executed) return ProposalState.Executed;
    if (block.timestamp < proposal.votingStartsAt) return ProposalState.Pending;
    if (block.timestamp <= proposal.votingEndsAt) return ProposalState.Active;

    // After voting: check quorum and approval
    if (!_meetsQuorum(proposalId) || !_meetsApproval(proposalId)) {
        return ProposalState.Defeated;
    }

    return ProposalState.Succeeded;
}
```

Impact

- `getProposal` will treat all proposals that meet the quorum and approval thresholds as successful, even though only the winner can actually be executed.
- After the winner of a cycle has been executed, if `_state` iterates over a proposal that is not the current cycle's winner but still meets the approval and quorum thresholds, `_checkNoExecutableProposals` will always assume there are still

executable proposals, even though there aren't. This currently has no impact, since execution always triggers the start of a new cycle.

Code Snippet

https://github.com/sherlock-audit/2025-11-levr-nov-4th/blob/main/levr-sc/src/LevrGovernor_v1.sol#L241-L252

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.