hashlock.

Security Audit

Blockstreet (DeFi)

August 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

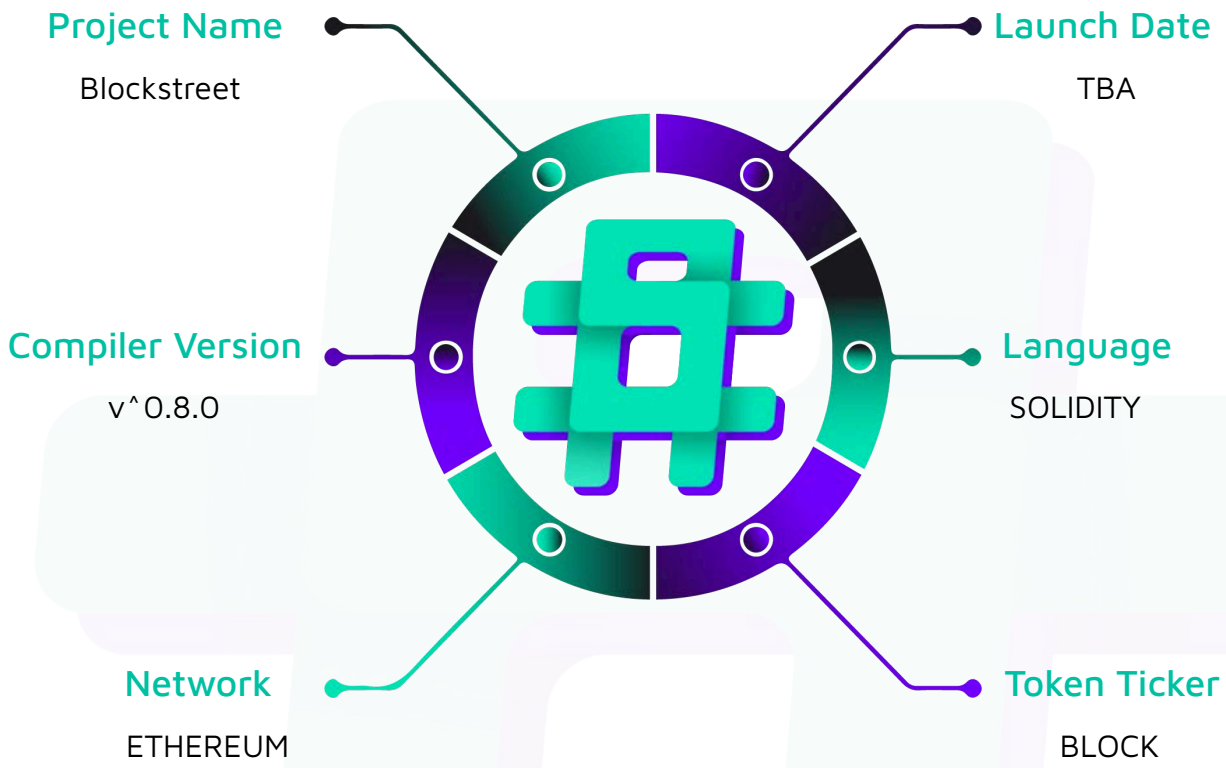
The Blockstreet team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

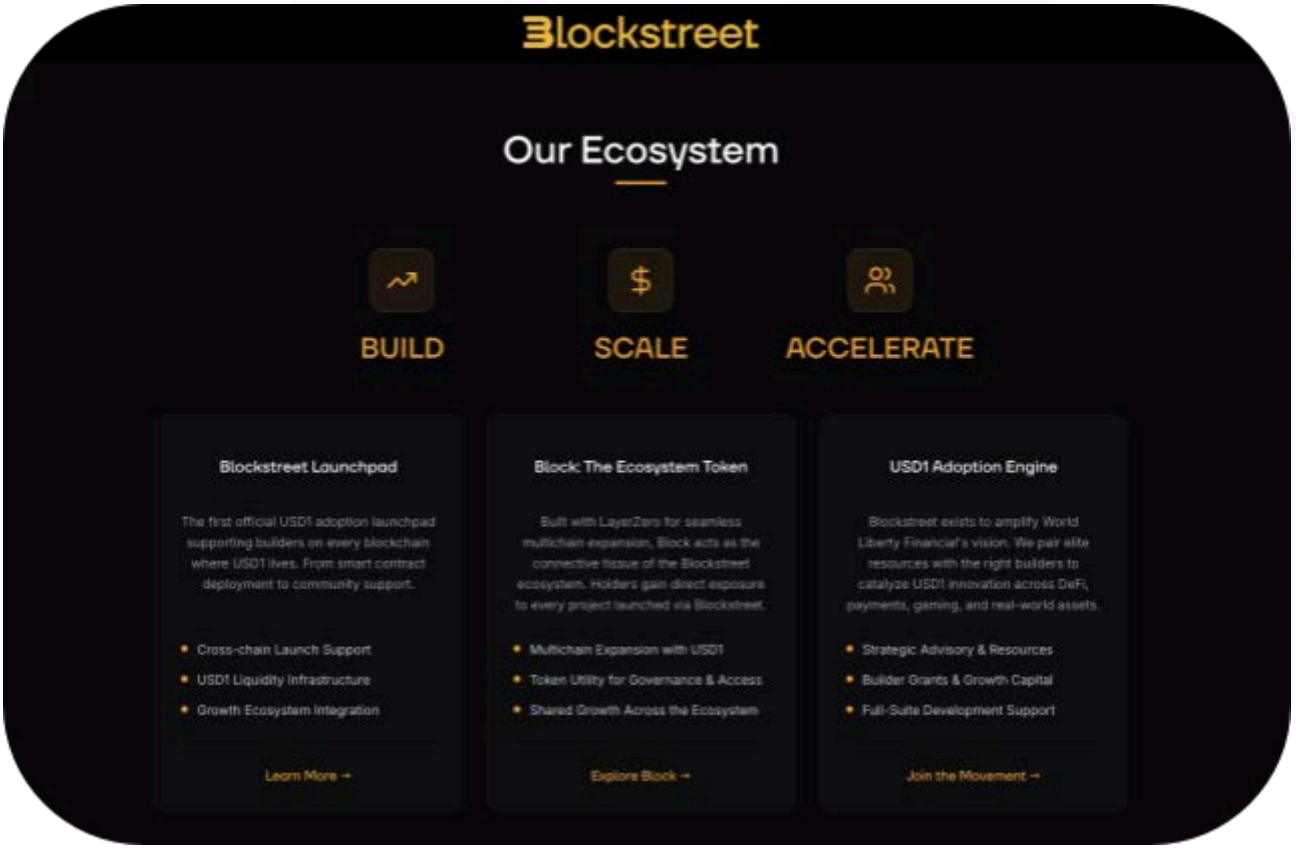
Blockstreet is a multichain platform launched with the goal of driving the adoption of USD1, a regulated stablecoin created by World Liberty Financial, serving as a hub for developers and projects in sectors such as DeFi, payments, gaming, and real-world assets (RWA). Built on LayerZero technology, the platform provides infrastructure for project launches (launchpad), USD1 liquidity, cross-chain support, community governance, and modular compliance tools (KYC/AML) to foster institutional adoption and innovation within the traditional-finance Web3 ecosystem. The native token BLOCK, with a maximum supply capped at 1 billion, is designed for utility, governance, and revenue sharing with holders, while also representing direct exposure to all projects created on the platform.

Project Name: Blockstreet
Project Type: DeFi
Compiler Version: ^0.8.0
Website: https://www.blockstreet.xyz/
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the Blockstreet project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Blockstreet Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**August, 2025**
Contract 1	Locked.sol
Contract 2	launchpad_v2.sol
Audited GitHub Commit Hash 1	f6eb237763a4ffb48ca5526cee78836c875f8451
Audited GitHub Commit Hash 2	7ea4c80720022018188b78f75c546cfae12c5c15
Fix Review GitHub Commit Hash	d3d0cdf4384afa9bb73c92fe5265b71f7321536e

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

3 Medium severity vulnerabilities

8 Low severity vulnerabilities

3 Gas Optimisations

1 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour

Locked.sol
- Allows users to:
  - Stake ERC20 tokens for specified lock periods
  - Earn rewards based on lock duration and amount
  - Withdraw staked tokens and rewards after lock period
  - Reset their stake to compound rewards
- Allows admins to:
  - Set reward rates for different lock periods
  - Recover ERC20 tokens from the contract
	Contract achieves this functionality.

launchpad_v2.sol
- Allows users to:
  - Participate in token launchpad rounds
  - Claim distributed tokens according to release schedules
  - View allocation, round, and wallet information
- Allows admins to:
  - Configure rounds, tiers, and release schedules
  - Withdraw raised or distributed tokens
  - Set key parameters (rate, cap, open/close times)
	Contract achieves this functionality.

Code Quality

This audit scope involves the smart contracts of the Blockstreet project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Blockstreet project smart contract code in the form of Github access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] StakingReward - Implementation contract can be maliciously initialized

Description

The contract is upgradeable but the implementation contract's initializer can be called by anyone if deployed without immediate initialization. An attacker could call initialize on the implementation contract (not the proxy) to become the owner and gain admin privileges. This is a common vulnerability in upgradeable contracts where the implementation is left uninitialized.

Vulnerability Details

The initialize function is marked as public and uses the initializer modifier. This ensures that the proxy contract cannot be initialized multiple times, but the implementation contract itself remains unprotected. If the implementation contract is deployed without constructor-level initialization, it is possible for an attacker to directly call initialize on the logic contract and assign ownership or roles to themselves.

```
contract StakingReward is
ReentrancyGuardUpgradeable,OwnableUpgradeable,AccessControlUpgradeable {

    ...

    function initialize(address _stakingToken) public initializer{

         stakingToken = IERC20Upgradeable(_stakingToken);

         __Ownable_init();

         __ReentrancyGuard_init();

         __AccessControl_init();

    }

    ...
```

```
}
```

Since there is no constructor safeguard in the logic contract, the implementation contract can be initialized by anyone, leaving it in a tainted state.

Proof of Concept

The following test demonstrates how an attacker can initialize the implementation contract directly:

```
function testImplementationInitializeTakeover() public {

    // malicious token the attacker will set

    address maliciousToken = address(0xdead);



    // attacker can initialize the implementation directly and hijack ownership

    vm.prank(hacker);

    StakingReward(stakingRewardImpl).initialize(maliciousToken);



    // asserts: attacker is now owner of the implementation,

    // and arbitrary stakingToken was set by the attacker

        assertEq(StakingReward(stakingRewardImpl).owner(), hacker, "owner should be hijacked");

      assertEq(address(StakingReward(stakingRewardImpl).stakingToken()), maliciousToken, "stakingToken should be attacker-controlled");

}
```

Impact

An attacker can become the owner of the implementation contract, potentially affecting future upgrades or causing confusion. While this doesn't directly affect the proxy's storage, it's a security best practice violation.

Recommendation

Add a constructor to the implementation contract that automatically marks it as initialized, preventing any external entity from calling initialize directly:

```
constructor() initializer {}
```

This ensures that only the proxy can perform initialization, and the logic contract itself cannot be hijacked.

Status

Resolved

[M-02] LaunchPad - Missing return value checks for ERC20 transfers

Description

The LaunchPad contract does not check the return values of critical ERC20 operations such as `transfer` and `transferFrom`. Certain tokens (e.g., USDT, BNB in older implementations) return false instead of reverting when a transfer fails. Without proper checks, the contract may continue execution even though the transfer has failed, causing severe accounting inconsistencies.

Vulnerability Details

When users participate, `transferFrom` may fail silently but the contract still records their contribution, inflating participation records. During claims, `transfer` may also fail silently, yet the contract marks the allocation as claimed, preventing users from retrying. Furthermore, a malicious actor could exploit this by supplying a non-standard ERC20 token that always returns false, allowing them to bypass actual transfers while still being registered as a participant.

Impact

Users can lose funds permanently if transferFrom() fails but participation is still recorded. During claims, users' allocations are marked as claimed even if transfer() fails, preventing them from ever claiming their tokens again. Malicious users could also participate without actually sending tokens if using a custom token that returns false.

Recommendation

Use OpenZeppelin's SafeERC20 library (safeTransfer, safeTransferFrom) to ensure all token transfers revert on failure.

Status

Resolved

[M-03] LaunchPad#_claim - Reentrancy Vulnerability with ERC777 Tokens Allows Unlimited Claims

Description

The _claim function in the LaunchPad contract is vulnerable to a reentrancy attack if the release token is an ERC777 token (via tokensReceived hook). This is because the function transfers tokens to the claimant before updating the claimed state, and ERC777 tokens can invoke hooks (such as tokensReceived) that allow the recipient to re-enter the contract and call claim again before the state is updated.

Vulnerability Details

In the _claim function, the contract performs the token transfer to the claimant before updating the claimed amount:

```
function _claim(uint256 index, address claimant) internal {


  require(_releaseToken != address(0), "Release token not yet defined");

  require(index < _releases.length, "Invalid release index");




  for (uint256 i = 0; i <= index; i++) {

      (,,,,,, uint256 remaining, uint256 status) = _getAllocation(claimant, i);


      if (status != 1) continue;


      // transfer may return false in some tokens; original logic ignored it

      IERC20(_releaseToken).transfer(claimant, remaining);




      _releases[i].claimed[claimant] = _releases[i].claimed[claimant] + remaining;
```

```
        emit Claimed(i + 1, claimant, remaining);
    }
}
```

If `_releaseToken` is an ERC777 token, the transfer call can trigger a reentrancy via the tokensReceived hook. Since the claimed state is only updated after the transfer, an attacker can repeatedly call claim (directly or via the hook) and drain the contract's tokens.

Impact

A malicious user can drain all available release tokens from the contract if the release token is ERC777 or any token with a reentrancy vector, resulting in a total loss of distributed tokens.

Recommendation

Update the claimed state before transferring tokens to the claimant.

```
function _claim(uint256 index, address claimant) internal {


    require(_releaseToken != address(0), "Release token not yet defined");
    require(index < _releases.length, "Invalid release index");



    for (uint256 i = 0; i <= index; i++) {
        (,,,,,, uint256 remaining, uint256 status) = _getAllocation(claimant, i);


        if (status != 1) continue;
```

```
+     _releases[i].claimed[claimant] = _releases[i].claimed[claimant] + remaining;



      // transfer may return false in some tokens; original logic ignored it
      IERC20(_releaseToken).transfer(claimant, remaining);



-     _releases[i].claimed[claimant] = _releases[i].claimed[claimant] + remaining;



      emit Claimed(i + 1, claimant, remaining);
  }
}
```

Status

Resolved

Low

[L-01] StakingReward#revokeAdminRole - Admins can revoke other admins

Description

In the StakingReward contract, the revokeAdminRole function can be called by any address with the onlyAdmins modifier. This creates a problem because an admin can revoke other admins (or even themselves).

```
function revokeAdminRole(address admin) public onlyAdmins {

    revokeRole(ADMIN_ROLE, admin);

}
```

Recommendation

Restrict this function so that only the owner can revoke admin roles.

Status

Resolved

[L-02] StakingReward#exit - Potential lack of stakingToken liquidity for payouts

Description

In the StakingReward contract, after the staking period ends, users can call exit to withdraw both their staked amount and earned rewards. The function transfers amount = staked + rewards directly from the contract.
If the contract does not hold enough stakingToken, the transfer will fail, and users may not understand the reason for the revert.

```
function exit(uint256 time) external  nonReentrant  {

    uint256 stakedTime = block.timestamp.sub(userLastStackedTime[_msgSender()][time]);


    uint256 _balanceTier =  _balancesTier[_msgSender()][time];


    require(stakedTime > time, "Withdraw locked");

    require(_balanceTier > 0, "Balance is 0");


    uint256 amount = _balanceTier.add(earned(_msgSender(),time));


    _totalSupply = _totalSupply.sub(_balanceTier);

    supply[time] = supply[time].sub(_balanceTier);


    _balances[_msgSender()] = _balances[_msgSender()].sub(_balanceTier);


    _balancesTier[_msgSender()][time] = 0;


    stakingToken.safeTransfer(_msgSender(), amount);

    emit Withdrawn(_msgSender(), amount);
```

```
}
```

Recommendation

Provide a clear and specific revert message when the contract lacks sufficient stakingToken balance, so users understand the failure reason.

Status

Resolved

[L-03] StakingReward#updateLockRate - Missing event for critical reward parameter

Description

The `updateLockRate` function modifies critical protocol parameters (reward rates) but doesn't emit an event. This makes it impossible to track rate changes on-chain, hiding potentially malicious admin actions from users and monitoring systems. Users cannot detect when rates are changed, making it difficult to verify if they're receiving expected rewards.

```
function updateLockRate(uint256 _lockedTime,uint256 _rewardRate) external onlyAdmins{

    lockRate[_lockedTime] = _rewardRate;

}
```

Recommendation

Emit an event whenever the reward parameter is updated.

Status

Resolved

[L-04] LaunchPad#setupReleases - Missing proper input validation

Description

The setupReleases function does not validate critical input conditions. First, it does not check that the sum of all release percentages equals 10000 (100%). If the total is less than 10000, some tokens may remain permanently locked, and if greater than 10000, users could claim more tokens than intended, potentially draining the contract.

Second, there is no validation that fromTimestamp < toTimestamp for each release. Without this check, a release could be misconfigured with invalid time ranges, leading to logical errors in reward distribution and claim periods.

Third, the function does not check that each fromTimestamp is properly aligned with the launchpad's lifecycle. If a release is configured with a fromTimestamp earlier than the launchpad's closeTimestamp(), users may be able to claim rewards while the round is still open and participation is ongoing. This overlap can create inconsistencies where users are both allowed to participate and claim simultaneously, leading to unfair or unintended behavior.

```
function setupReleases(
    uint256[] calldata fromTimestamps,
    uint256[] calldata toTimestamps,
    uint256[] calldata percents
) external onlyOwner {
    require(fromTimestamps.length == toTimestamps.length, "Invalid releases");
    require(toTimestamps.length == percents.length, "Invalid releases");

    delete _releases;

    for (uint256 i = 0; i < fromTimestamps.length; i++) {
        _releases.push();
```

```
        Release storage r = _releases[i];

        r.fromTimestamp = fromTimestamps[i];

        r.toTimestamp = toTimestamps[i];

        r.percent = percents[i];

    }

}
```

Recommendation

Improper release validation can cause tokens to be permanently locked, over-distributed, or claimable while participation is still ongoing. This breaks the expected vesting logic and undermines fairness and reliability of the fundraising process.

Status

Resolved

[L-05] LaunchPad#releaseTokenDecimals - Returns 18 even if release token is not set, leading to potential calculation errors

Description

In the LaunchPad contract, the releaseTokenDecimals function is used to determine the decimals of the release token for allocation and claim calculations. If the release token is not set (_releaseToken == address(0)), the function simply returns 18 by default, rather than reverting or signaling an error.

```
function releaseTokenDecimals() public view returns (uint256 decimals) {
    decimals = 18;
    if (_releaseToken != address(0)) {
        decimals = IERC20(_releaseToken).decimals();
    }
    return decimals;
}
```

Recommendation

Change the function to revert if _releaseToken is not set, for example:

```
            function releaseTokenDecimals() public view returns (uint256 decimals) {
+               require(_releaseToken != address(0), "Release token not set");
-               decimals = 18;
+               decimals = IERC20(_releaseToken).decimals()
-               if (_releaseToken != address(0)) {
-                   decimals = IERC20(_releaseToken).decimals();
-               }
                return decimals;
            }
```

Status

Resolved

[L-06] LaunchPad#participate - Does not revert when msg.value is sent with ERC20 participation, leading to potential user fund loss

Description

In the LaunchPad contract, the participate function allows users to participate using either the native token or an ERC20 token. However, when participating with an ERC20 token (_raiseToken != address(0)), the function does not revert if a non-zero msg.value is sent. This can result in users accidentally sending native tokens along with their ERC20 participation, causing those native tokens to be locked in the contract.

```
function participate(address token, uint256 amount) public payable {

    ...

    if (_raiseToken == address(0)) {

        require(msg.value == amount, "Insufficent native token sent to contract");

    } else {

        // No check for msg.value > 0 here

        IERC20 t = IERC20(_raiseToken);

        require(t.allowance(msg.sender, address(this)) >= amount, "Insufficent allowance,
approve first");

        t.transferFrom(msg.sender, address(this), amount);

    }

    ...

}
```

Recommendation

Add a check to revert the transaction if msg.value > 0 when participating with an ERC20 token

Status

Resolved

[L-07] LaunchPad#transferNativeToken - Use of transfer may cause reverts due to gas stipend limitations

Description

In the LaunchPad contract, the transferNativeToken function uses Solidity's transfer method to send native tokens to a specified address. The transfer method forwards a fixed 2300 gas stipend, which may not be sufficient for some recipient contracts, causing the transaction to revert unexpectedly.

```
function transferNativeToken(uint256 amount, address payable to) public onlyOwner {
    require(address(this).balance >= amount, "Insufficient native token balance");
    to.transfer(amount); // May revert if recipient requires more than 2300 gas
}
```

Recommendation

Use a low-level call pattern to forward all available gas and handle failures explicitly

Status

Resolved

[L-08] Contract - Outdated OpenZeppelin Version

Description

The contract uses an outdated version of the OpenZeppelin Ownable implementation. Using older versions may expose the contract to known bugs or miss out on security and gas optimizations present in newer releases.

Recommendation

Upgrade to the latest stable version of OpenZeppelin contracts.

Status

Resolved

Gas

[G-01] StakingReward - Unnecessary use of SafeMath in Solidity ^0.8.3

Description

The contract uses the SafeMathUpgradeable library despite being compiled with Solidity ^0.8.3, which has built-in overflow/underflow protection. This adds unnecessary gas overhead to every arithmetic operation without providing additional safety benefits.

```
contract StakingReward is
ReentrancyGuardUpgradeable,OwnableUpgradeable,AccessControlUpgradeable {

    using SafeMathUpgradeable for uint256;
```

Recommendation

Remove the SafeMathUpgradeable library and use native Solidity arithmetic operators instead.

Status

Acknowledged

[G-02] LaunchPad#transferToken - Unnecessary balance check before transfer, leading to increased gas usage

Description

In the LaunchPad contract, the transferToken function includes a redundant check to ensure the contract's ERC20 token balance is sufficient before calling transfer. The ERC20 transfer function will already revert if the balance is insufficient, making this explicit check unnecessary and increasing gas consumption.

```
function transferToken(address token, uint256 amount, address to) public onlyOwner {

    IERC20 t = IERC20(token);

    require(t.balanceOf(address(this)) >= amount, "Insufficient token balance"); //
Unnecessary

    t.transfer(to, amount);

}
```

Recommendation

Remove the redundant require(t.balanceOf(address(this)) >= amount, ...) statement to save gas, as the ERC20 transfer will revert on insufficient balance.

Status

Acknowledged

[G-03] Contract - Public Function Not Used Internally

Description

Public functions that are not called internally can be marked as external to save gas.

Recommendation

Change visibility from public to external.

Status

Acknowledged

QA

[Q-01] Contract - Public variable naming does not follow Solidity conventions

Description

In the StakingReward and LaunchPad contracts, several public variables are declared with an underscore (_) prefix. This does not follow Solidity's naming conventions and may reduce readability and clarity for auditors and users.

Recommendation

Remove the underscore prefix from public variables, or make them private and provide explicit getters if needed.

Status

Acknowledged

Centralisation

The Blockstreet project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the Blockstreet project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au