hashlock.

2025

Security Audit

BUX (Reward system for token holders)

November 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

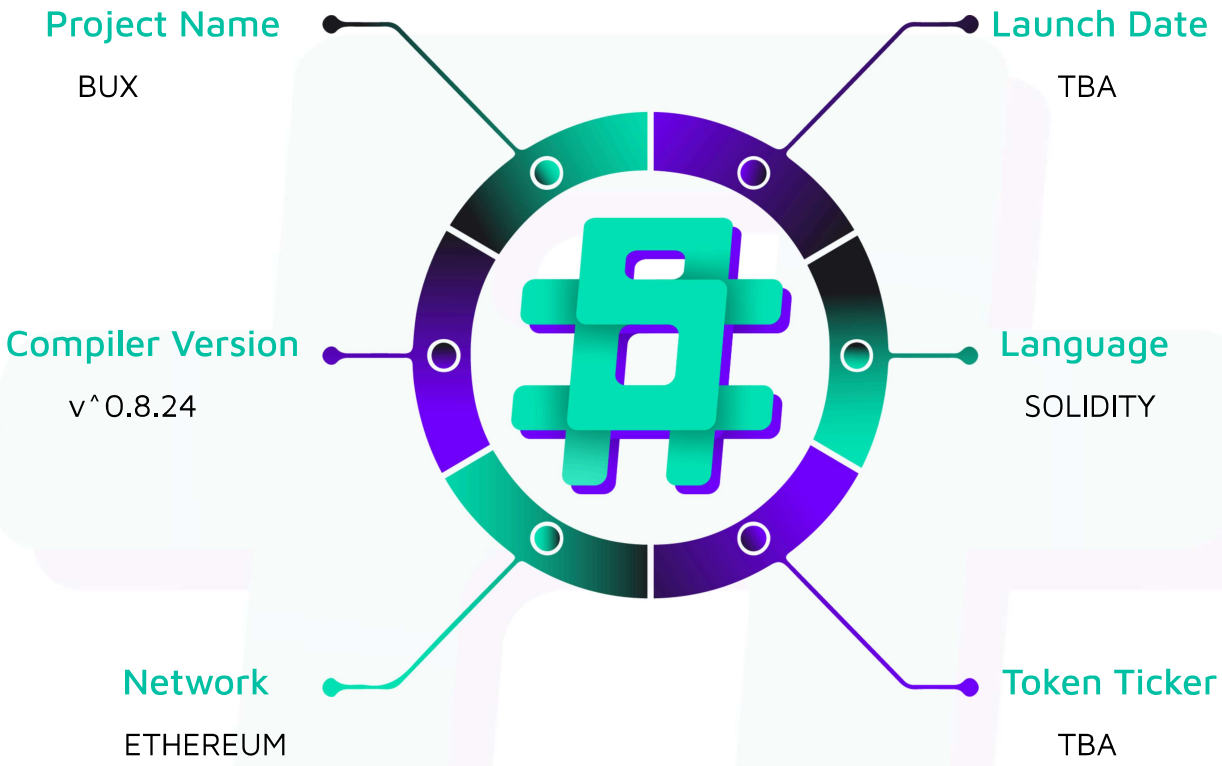
The BUX team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

The BUX Reward system for token holders is a decentralized and automated Reward system for token holders platform built on Ethereum, designed to reward users through ETH prize pools funded by trading activity on Uniswap v4. It leverages Chainlink VRF to ensure transparent and verifiable randomness and Chainlink Automation to execute scheduled draws. Token holders of BUX participate in weighted random selections based on their holdings, creating a fair and auditable Reward system for token holders mechanism. The website under development will likely serve as a user-facing interface for managing participation, viewing live draws, and tracking rewards in a trustless and self-custodial environment.

Project Name: BUX
Project Type: Reward system for token holders
Compiler Version: ^0.8.24
Website: TBA
Logo:
TBA

Visualised Context:



Audit Scope

We at Hashlock audited the solidity code within the BUX 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	**BUX Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**November, 2025**
Contract 1	BUXLotto.sol
Contract 2	BUXToken.sol
Contract 3	DevFeeSplitter.sol
Contract 4	LottoFundingHook.sol
Contract 5	SortitionIndex.sol

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.

Hashlock found:

4 High severity vulnerabilities

2 Medium severity vulnerabilities

2 Low severity vulnerabilities

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
BUXLotto.sol - Automated ETH Reward system for token holders with hourly and daily draws funded by Uniswap v4 trading fees. - Manages two independent Reward system for token holders draws: hourly (with minimum pot size, resets hourly) and daily (no minimum, resets every 24 hours). - Receives ETH from a Uniswap v4 hook via fundFromHook() to fund prize pots. - Uses Chainlink Automation (performUpkeep()) and Chainlink VRF (requestRandomWords(), rawFulfillRandomWords()) for random winner selection via SortitionIndex. - Includes ReentrancyGuard, zeroes pot amounts before VRF requests, rolls back prizes on failed transfers, has timeout recovery for stuck VRF requests, and owner-controlled pausing. - Requires registration as a VRF consumer, Chainlink Automation upkeep, authorized hook, and deployed SortitionIndex and BUXToken contracts.	**Contract achieves this functionality.**
BUXToken.sol - ERC20 token for BUX Reward system for token holders with weighted random selection. - Eligibility based on minEligibleBalance for EOAs, with contracts needing explicit allowlisting. - Integrates with SortitionIndex to update per-account weights during transfers, mints, and burns. - Supports pausing of transfers, with exceptions for owner and pause-exempt addresses. - Ensures initialRecipient is a contract and uses Ownable2Step for secure ownership transfers.	**Contract achieves this functionality.**

DevFeeSplitter.sol - Distributes development team fees using a pull-based payment pattern. - Employs a pull pattern to prevent reentrancy during funding, isolate recipient claim failures, and avoid griefing attacks. - ETH arrives via receive() or fund() and is processed upon claim: unaccounted ETH is split by sharesBps, and claimableEth balances are updated. - Recipient shares and remainderSinkIndex can be updated by the owner (unless frozen), with shares summing to 10,000 basis points. Configuration can be permanently frozen. - Includes Pausable functionality to stop claims, emergencyWithdraw for the owner to recover unaccounted funds when paused, and Ownable2Step for safe ownership transfer.	**Contract achieves this functionality.**
LottoFundingHook.sol - A Uniswap v4 hook that captures fees from BUX/ETH swaps to fund the Reward system for token holders. - Intercepts BUX/ETH swaps to extract fees, requiring specific permission flags (0x00CC) in its address for beforeSwap and afterSwap functionality. - Extracts a percentage of ETH from swaps - ETH/BUX swap fees are collected before/after, always from the user.	**Contract achieves this functionality.**
SortitionIndex.sol - Weighted random selection uses O(log N) Fenwick tree. - "Soft deletion" keeps zero-weight accounts, prevents array changes. - Controller modifies weights, drawByUint avoids zero-weight, Ownable2Step security. - Owner can compact index, removing soft-deleted entries in chunks. - Batch mode supported for bulk weight updates (off-chain monitoring).	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the BUX 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 were 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 BUX project smart contract code in the form of ZIP 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

High

[H-01] BUXLotto#setCallbackGasLimit - Unconstrained Callback Gas Limit

Description

The `setCallbackGasLimit` function allows the owner to configure the `Chainlink VRF` callback gas limit without a lower bound, leading to a critical operational denial of service and potential fund inaccessibility.

Vulnerability Details

The `_vrf.callbackGasLimit` parameter, crucial for `Chainlink VRF` fulfilment, is directly set via `newGas` (`Line 218`) without any lower bound validation. If newGas is set below the actual gas required for `rawFulfillRandomWords()` (`Line 463`) and its internal settlement logic (`_settleHourly()/_settleDaily()`) to complete, the `VRF` callback will run out of gas and revert. This prevents the clearing of pending draw states, causing `performUpkeep()` (`Lines 320, 345`) to halt new draws for that type, leading to a complete operational denial of service. Additionally, prize funds for stuck draws become inaccessible through the automated system, requiring manual owner intervention.

```
function setCallbackGasLimit(uint32 newGas) external onlyOwner {
        uint32 old = _vrf.callbackGasLimit;
        _vrf.callbackGasLimit = newGas;
        emit CallbackGasLimitUpdated(old, newGas);
    }
```

An unconstrained callback gas limit can halt draws, lock funds operationally, and require manual owner recovery.

Impact

Operational Denial of Service for Reward system for token holders draws and potential inaccessibility of prize funds until manual owner recovery.

Recommendation

We recommend implementing a strict require statement to enforce a safe lower bound on the `newGas` parameter in the `setCallbackGasLimit` function. This

`MIN_SAFE_CALLBACK_GAS_LIMIT` should be an immutable constant, carefully calculated to guarantee successful VRF callback execution.

Status

Resolved

[H-02] BUXLotto#rescueETH - Critical Accounting Inconsistency and Phantom Funds

Description

The `rescueETH` function allows the owner to withdraw `ETH` without updating internal prize pot state variables, leading to severe accounting discrepancies and a complete operational halt of the Reward system for token holders.

Vulnerability Details

The `rescueETH` function successfully transfers amount `ETH` from the contract's actual balance (`address(this).balance`) to a specified recipient (`Line 621`). However, it critically fails to decrement the internal `hourlyPotWei` or `dailyPotWei` state variables (`Lines 109-110`). This breaks the fundamental financial invariant, causing the internal pot records to report "phantom funds" that the contract no longer possesses. Consequently, `performUpkeep()` (Line 315) attempts to initiate draws based on these inflated internal balances, leading to `payable(winner).call` failures during `_settleHourly()/_settleDaily()` (`Lines 527, 570`). These failures cause Chainlink VRF fulfillments to revert, leaving draws stuck and ultimately resulting in a complete operational denial of service for the Reward system for token holders's core functionality, further exacerbated by incorrect "`rollback`" additions of phantom funds.

```
function rescueETH(uint256 amount, address payable to) external onlyOwner whenPaused {
        if (to == address(0)) revert ZeroAddress();
        (bool success, ) = to.call{value: amount}("");
        require(success, "rescue failed");
    }
```

Draining `ETH` via `rescueETH` breaks accounting, creating phantom funds, and halts all Reward system for token holders operations.

Impact

Complete operational denial of service for Reward system for token holders draws, fundamental breach of financial invariant, guaranteed transaction failures for prize payouts, and misleading contract state.

Recommendation

We recommend modifying the `rescueETH` function to explicitly update and decrement the `hourlyPotWei` and/or `dailyPotWei` state variables by the amount of `ETH` withdrawn. This ensures that the contract's internal accounting accurately reflects its actual `ETH` balance. Ideally, the function should require the owner to specify which pot the funds

are being rescued from, or implement a clear strategy for proportional deduction, maintaining the invariant `address(this).balance >= hourlyPotWei + dailyPotWei`.

Status

Resolved

[H-03] DevFeeSplitter#pause()/unpause() - Owner Can Indefinitely Lock User Claims

Description

The contract owner possesses exclusive control over pausing and unpausing the contract, effectively granting the power to indefinitely prevent users from accessing their claimable funds.

Vulnerability Details

The pause mechanism creates a centralization risk, as only the owner can unpause the contract. If the owner becomes inactive, loses access, or acts maliciously, the contract could remain paused indefinitely. This would effectively prevent users from withdrawing funds and introduces a single point of failure that compromises system reliability.

```
function pause() external onlyOwner {
   _pause();
      emit Paused(); }
  function unpause() external onlyOwner {
      _unpause();
      emit Unpaused();}
```

The `onlyOwner` modifier on both `pause()` and `unpause()` gives the owner ultimate control over users' access to funds.

Impact

Users are entirely dependent on the contract owner to regain access to their funds after a pause. This introduces significant centralization risk, as a malicious, inactive, or compromised owner can permanently prevent users from claiming, leading to a denial of access to funds and a breakdown of trust in the contract's functionality.

Recommendation

We recommend considering a timelock mechanism for the `pause()` and/or `unpause()` functions, or a multi-signature wallet for ownership, to reduce reliance on a single entity's discretion.

Status

Resolved

[H-04] LottoFundingHook#afterSwap - Double Fee Collection for ETH→BUX (Specified Input)

Description

For ETH→BUX swaps with specified `ETH` input, fees are charged twice. `_beforeSwap` correctly captures the fee, but `_afterSwap` erroneously re-extracts and forwards the same fee from the pool.

Vulnerability Details

The contract's logic intends to capture fees for `ETH→BUX` swaps with specified `ETH` input exclusively in `_beforeSwap` by returning a `BeforeSwapDelta` to increase the user's input. However, the `_afterSwap` function, specifically in its else block (`lines 374-385`) corresponding to this scenario, mistakenly recalculates this same fee and then calls `poolManager.take()` to extract it again from the Uniswap pool, subsequently forwarding it via `_forwardSplitAndEmit()`. This results in the user effectively paying the fee twice for the same swap: once as increased input, and again from the pool's liquidity.

```
  (within _afterSwap function)     }
 else {
           uint256 ethMoved = _abs(params.amountSpecified);
uint256 fee = _computeFee(ethMoved);
          if (fee > 0) {
              poolManager.take(_cEth, address(this), fee);
               _forwardSplitAndEmit(poolId, true, ethMoved, fee); }
```

This block attempts to take and forward fees again, `contradicting _beforeSwap's` role.

Impact

Users performing `ETH→BUX` swaps with specified `ETH` input will incur direct and significant financial loss, being charged double the intended fee. This also leads to an inflated and incorrect accumulation of fees for the Reward system for token holders and dev splitter.

Recommendation

We recommend removing lines `379-382` from the `_afterSwap` function's else block. This specific section, containing `poolManager.take()` and `_forwardSplitAndEmit()`, should be entirely omitted as the fee for this swap type is handled by `_beforeSwap`.

Status

Resolved

Medium

[M-01] BUXLotto#setMinHourlyPrizeWei - Unconstrained minHourlyPrizeWei Parameter

Description

The `setMinHourlyPrizeWei` function allows the contract owner to set the minimum prize for hourly draws without any upper bound, posing a critical operational denial of service risk.

Vulnerability Details

The absence of an upper bound for `minHourlyPrizeWei` allows the owner to set an excessively high value, either intentionally or accidentally, preventing hourly draws from ever triggering. This results in a denial of service for the hourly Reward system for token holders and also blocks the `recoverStuckDraw()` function, as its re-request condition would consistently fail.

```
function setMinHourlyPrizeWei(uint256 newMin) external onlyOwner {
       uint256 old = minHourlyPrizeWei;
       minHourlyPrizeWei = newMin;
       emit MinHourlyPrizeUpdated(old, newMin);}
```

Unconstrained minimum prize enables denial-of-service, halting hourly draws and impeding recovery mechanisms.

Impact

Complete operational denial of service for hourly Reward system for token holders draws, preventing new draws from starting and potentially hindering recovery of stuck draws.

Recommendation

We recommend implementing a strict upper bound check on the newMin parameter in the `setMinHourlyPrizeWei` function. This upper limit should be an immutable constant, `MAX_REALISTIC_HOURLY_PRIZE_WEI`, to prevent economically unrealistic or operationally detrimental values from being set.

Status

Resolved

[M-02] BUXLotto#setDrawTimeout - Unconstrained Draw Timeout Parameter

Description

The setDrawTimeout function allows the contract owner to set the timeout for stuck VRF requests without an upper bound, leading to prolonged operational denial of service and temporary fund inaccessibility.

Vulnerability Details

The drawTimeout parameter, which defines the window for recovering stuck `VRF` requests, is directly set via newTimeout (`Line 245`) without any upper bound validation. If `newTimeout` is configured to an excessively large value, the `recoverStuckDraw()` function's `TimeoutNotReached()` check (`Line 402`) will perpetually revert. This critical failure in the recovery mechanism prevents the hourlyPending or `dailyPending` flags from being reset, thereby halting `performUpkeep()` (`Lines 320, 345`) from initiating any new draws. This results in a prolonged operational denial of service for the Reward system for token holders and renders associated prize funds operationally inaccessible for the duration of the artificially extended timeout.

```
function setDrawTimeout(uint64 newTimeout) external onlyOwner {
        if (newTimeout == 0) revert InvalidRequest();
        uint64 old = drawTimeout;
        drawTimeout = newTimeout;
        emit DrawTimeoutUpdated(old, newTimeout);
    }
```

Unconstrained timeout cripples recovery, leading to prolonged Reward system for token holders `DoS` and inaccessible prize funds.

Impact

Prolonged Operational Denial of Service for Reward system for token holders draws and temporary inaccessibility of prize funds until manual owner intervention and a waiting period.

Recommendation

We recommend implementing a strict require statement to enforce a safe upper bound on the newTimeout parameter in the `setDrawTimeout` function. This

`MAX_SENSIBLE_DRAW_TIMEOUT` should be an immutable constant, carefully determined by the client based on their operational risk tolerance and expected recovery timelines.

Status

Resolved

Low

[L-01] DevFeeSplitter#accrue() - DoS due to Unbounded Loop

Description

Several core functions in `DevFeeSplitter`, including `fund()`, `claim()`, `claim(uint256 amount)`, `claimTo(address payable to)`, and `payoutMany(address[] calldata recipientsToPay)`, are critically vulnerable to a `denial-of-service` (`DoS`) attack. This occurs if the number of configured recipients is excessively large, due to an unbounded loop within the internal `_accrue()` function, which all these functions indirectly call via `_accrueNewFunds()`.

Vulnerability Details

Functions like `fund()`, `claim()` (`lines 115 and 120`), `claimTo()` (`line 125`), and `payoutMany()` (`line 130`) all directly or indirectly call _accrueNewFunds(). This function, in turn, executes `the _accrue()` function (`line 269`). Inside _accrue(), a for loop (`lines 281-289`) iterates through the entire _recipients array to distribute funds. Each iteration involves expensive SSTORE operations (updating claimableEth balances) and event emissions. The _setRecipients function (lines 227-249), which manages this array, critically lacks any upper limit check on its length. Consequently, if a malicious or negligent owner configures an overly large number of recipients, the cumulative gas cost of _accrue() will exceed the Ethereum block gas limit, causing any transaction that calls any of the affected functions (fund, claim, claimTo, payoutMany) to deterministically revert with an "out of gas" error and permanently locking all funds.

```
for (uint256 i = 0; i < len; ++i) {
    uint256 share = (amount * _sharesBps[i])
    distributed += share;
    if (share > 0) {
        address recipient = _recipients[i];
        claimableEth[recipient] += share;
        emit Accrued(recipient, share);
    }
}
```

If the recipients list grows too large, this loop makes all related functions unexecutable.

Impact

This leads to a complete denial of service for multiple core contract functionalities. New funds cannot be processed, and all recipients are prevented from withdrawing their

accumulated Ether, effectively locking all funds indefinitely and rendering the contract unusable.

Recommendation

We recommend implementing a strict maximum limit for the number of recipients in the `_setRecipients` function. This requires defining a `uint256` public constant `MAX_RECIPIENTS` and adding a check to ensure that the length of the recipients_ array does not exceed this constant before updating the state. This will prevent the _accrue() loop from becoming excessively gas-intensive and causing DoS.

Status

Resolved

[L-02] LottoFundingHook#forwardSplitAndEmit - Fee Distribution Coupling / Single Point of Failure

Description

The `_forwardSplitAndEmit` function couples fee distributions to lotto and devFeeSplitter. If the `lotto.fundFromHook()` external call fails, the entire transaction reverts, preventing any fees from being distributed.

Vulnerability Details

In the `_forwardSplitAndEmit` function (`lines 437-475`), the hourlyAmt + dailyAmt is sent to the lotto contract via an external call to `lotto.fundFromHook()` (`lines 455-458`). If this call fails (e.g., lotto contract reverts due to a bug, misconfiguration, or temporary gas issues), line 458 `(if (!ok1) revert TransferFailed();)` immediately reverts the entire `_forwardSplitAndEmit` transaction. Consequently, the devAmt portion, intended for devFeeSplitter (lines 462-465), is never forwarded. This means a failure in one recipient's contract completely blocks the distribution of fees to all recipients for that specific swap.

```
if (hourlyAmt + dailyAmt > 0) {
            (bool ok1, ) = lotto.call{value: (hourlyAmt + dailyAmt)}(
                        abi.encodeWithSelector(IBUXLotto.fundFromHook.selector, hourlyAmt,
dailyAmt)
            );
            if (!ok1) revert TransferFailed();
        }
        if (devAmt > 0) {
            (bool ok2, ) = devFeeSplitter.call{value: devAmt}(\"\");
            if (!ok2) revert TransferFailed();
        }
```

A failure at line `458` prevents the `devAmt` from being sent.

Impact

This creates a single point of failure. An issue within the lotto contract can stop all fee distribution, leading to lost revenue for both the Reward system for token holders and the dev team.

Recommendation

We recommend decoupling the payout successes to prevent a single recipient's failure from halting all distributions. Implement try/catch around the lotto.call in `_forwardSplitAndEmit` to log failures and allow the `devFeeSplitter` payout to proceed

independently. Alternatively, consider a full pull-based payment model for both recipients, completely isolating their withdrawal processes.

Status

Resolved

QA

[Q-01] All Contracts - Floating Pragma Version

Description

All contracts use a floating pragma `^0.8.30` instead of locking to a specific compiler version.

Vulnerability Details

The contracts use pragma solidity `^0.8.30`, which allows compilation with any version from `0.8.30` up to (but not including) `0.9.0`. This creates deployment inconsistencies where different compiler versions may produce different bytecode, introducing subtle bugs or behavioral changes across deployments.

```
pragma solidity ^0.8.26;
```

Impact

Different compiler versions may introduce bugs or behavioral differences. Testing on one version doesn't guarantee the same behavior when deployed with another version, potentially causing unexpected issues in production deployments.

Recommendation

Lock the pragma to a specific compiler version using pragma solidity `0.8.30`; to ensure consistent compilation and deployment behavior across all environments.

Status

Resolved

Centralisation

The BUX 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 BUX project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved. 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