hashlock.

Security Audit

MarketDAO (DeFi)

January 2026 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

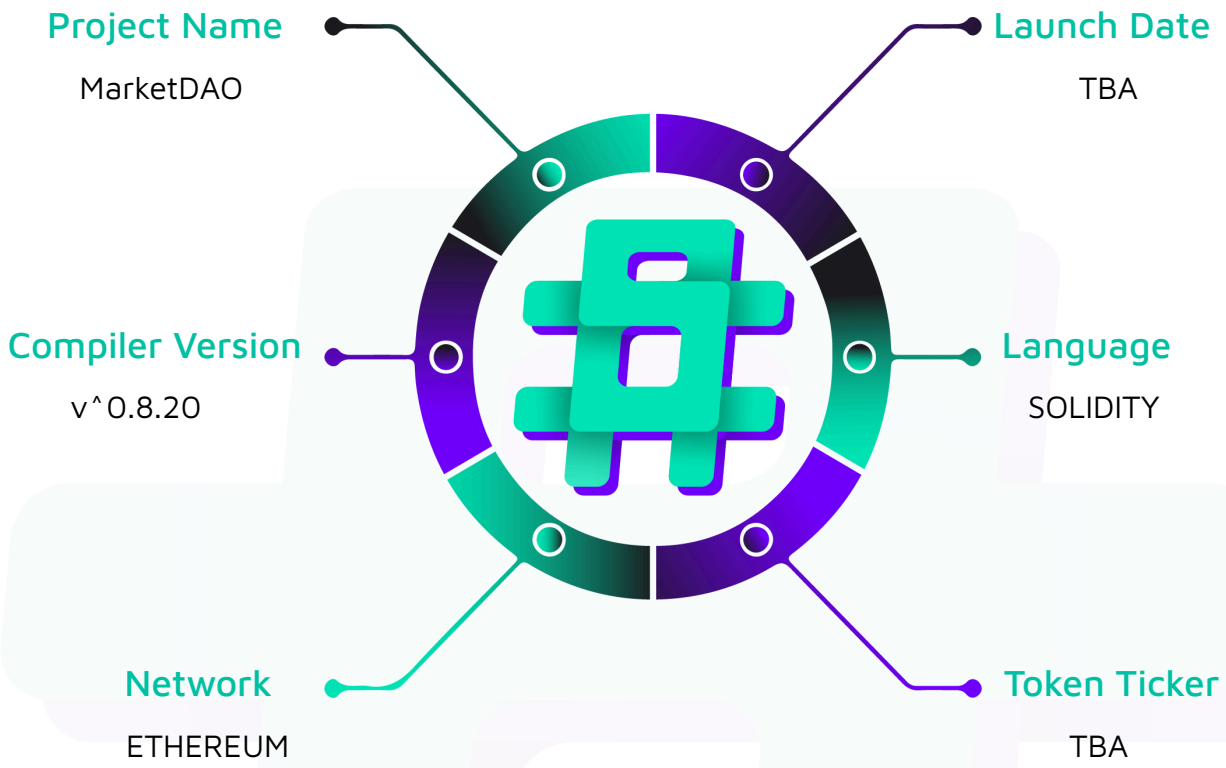
The MarketDAO team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

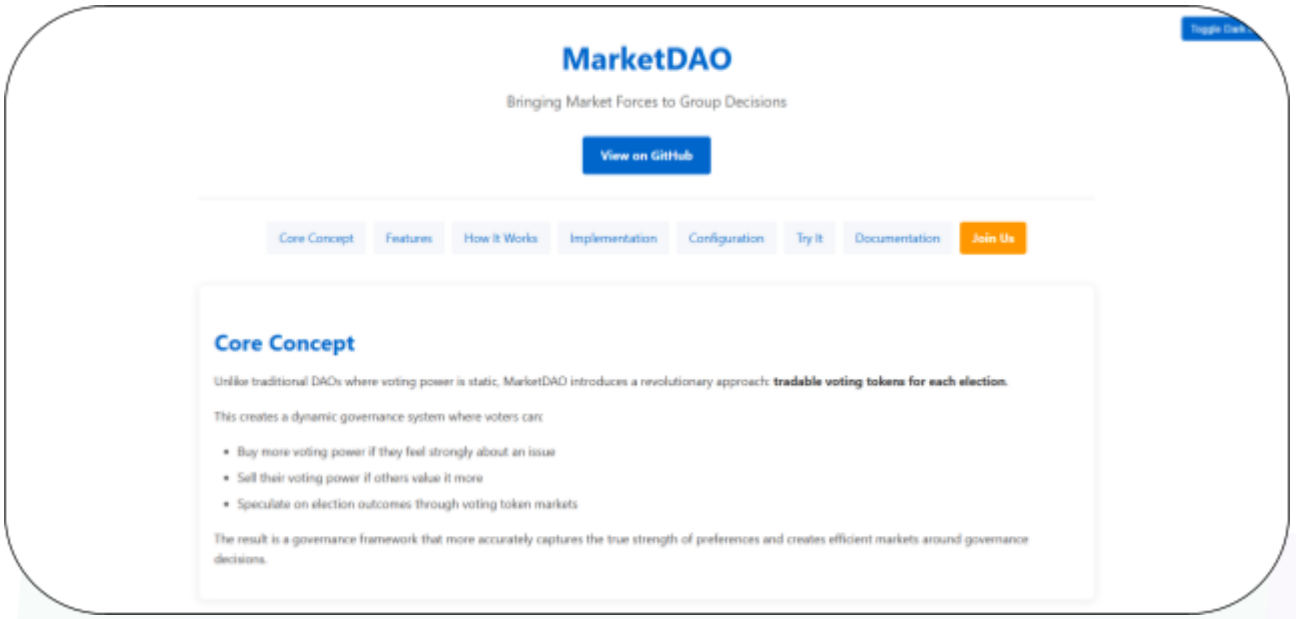
MarketDAO is a governance framework that brings market forces to bear on group decisions. The key innovation is a system where voting rights can be freely bought and sold during elections, allowing market forces to influence governance outcomes.

Project Name: MarketDAO
Project Type: Defi
Compiler Version: ^0.8.20
Website: https://marketdao.dev
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the MarketDAO 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	**MarketDAO Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**January, 2026**
Github 1	https://github.com/evronm/swapEscrow/tree/main
Contract 1	Escrow.sol
Contract 2	EscrowFactory.sol
Github 2	https://github.com/evronm/marketDAO
Contract 3	MarketDAO.sol
Contract 4	Proposal.sol
Contract 5	ProposalFactory.so
Contract 6	ProposalTypes.sol
Contract 7	DistributionRedemption.sol
Audited GitHub Commit Hash 1	668170459cdd53deac93c533fbc5d7876184eeed
Audited GitHub Commit Hash 2	a4722eb70a7efce0003dd7259a18aa14eadcbbd8
Fix Review GitHub Commit Hash 1	06df83d19c749a2bfdcc59481e2b9b95022fcb0c
Fix Review GitHub Commit Hash 2	4cce19e6b92e91f6eab3216af6ff25115be87721

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:

5 High severity vulnerabilities

3 Medium severity vulnerabilities

4 Low severity vulnerabilities

4 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
Escrow.sol - Single-use atomic-swap escrow - accepts deposits (ETH/ERC20/ERC721/ERC1155), - starts a timeout on first deposit, - detects payments via receive(), process() or token callbacks, - then on matching payment sends the payment to the first depositor and transfers all deposited assets to the payer; - allows original depositors to withdraw after expiry.	**Contract achieves this functionality.**
EscrowFactory.sol - Deploys cheap EIP-1167 clone instances of the Escrow implementation, - initializes each clone with the escrow parameters (duration, payment token/type/amount), - and records the created escrow addresses.	**Contract achieves this functionality.**
MarketDao-contracts - MarketDAO: ERC1155 DAO with governance token (id 0), vesting-gated transfers, and optional token sales/flags. - Proposals: collect support from vested holders; if support crosses threshold (of vested supply) it triggers an election before expiry. - Elections: mint a new voting token id, create	**Contract achieves this functionality.**

yes/no vote addresses, and let holders claim voting power from a snapshot at election start. - Passing rules: can early-execute on strict majority, else after duration requires quorum and yes > no; failures finalize and clear. - Treasury proposals: DAO can hold ETH/ERC20/ERC721/ERC1155; proposals lock funds on election and transfer on execution. - Factory: clones typed proposals (resolution/treasury/mint/parameter/distribution) and registers them as active in the DAO. - Distribution: deploys a redemption vault, transfers distribution funds to it on pass, and lets registered holders claim once pro-rata by recorded balance.	

Code Quality

This audit scope involves the smart contracts of the MarketDAO 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 MarketDAO 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

High

[H-01] Escrow#process() - Front-Running Allows Attacker To Misattribute Deposits

Description

The `process` function determines ownership and intent of newly received ERC20 tokens using a balance-delta check and a caller-supplied depositor, allowing any external account to race the legitimate user after a token transfer and control how the funds are attributed.

Vulnerability Details

The escrow expects users to transfer ERC20 tokens to the contract and then call `process` to account for the deposit or trigger a payment swap. However, `process` is externally callable by anyone and relies on `IERC20(token).balanceOf(address(this)) - processedERC20[token]` to infer the "new" amount.

```
function process(address token, address depositor) external {

        if (!initialized) revert NotInitialized();

        if (completed) revert EscrowAlreadyCompleted();

        if (expiryTime != 0 && block.timestamp >= expiryTime) revert EscrowExpired();

        // Check current balance vs processed balance

        uint256 currentBalance = IERC20(token).balanceOf(address(this));

        uint256 alreadyProcessed = processedERC20[token];

        if (currentBalance <= alreadyProcessed) revert NothingToProcess();

        uint256 newAmount = currentBalance - alreadyProcessed;

        // Mark as processed first (before swap to prevent reentrancy)
```

```
        processedERC20[token] = currentBalance;

        // Check if this matches payment parameters

        if (_isPayment(AssetType.ERC20, token, 0, newAmount)) {

            _executeSwap(depositor); // depositor here is actually the payer

        } else {

            // It's a deposit - add to locked assets

            _addDeposit(AssetType.ERC20, token, 0, newAmount, depositor);

        }

    }
```

An attacker can monitor the mempool, detect a victim's ERC20 transfer to the escrow, and front-run the victim's process call. By calling `process` first and supplying themselves as `depositor`, the attacker causes the escrow to record the victim's tokens as either a deposit or a payment attributed to the attacker, depending on `_isPayment`. The root cause is the combination of (a) balance-delta based accounting and (b) untrusted, user-supplied attribution (`depositor`) without binding it to the actual token sender.

Impact

Victim funds can be misattributed, swapped, or locked under attacker-controlled accounting, resulting in loss of funds, incorrect settlement, or denial of the intended escrow outcome.

Recommendation

We recommend modifying the `process` function to bind deposits to an authenticated sender or explicit transfer intent, for example by using pull-based transfers (`transferFrom`).

Status

Resolved

[H-02] DistributionRedemption#registerForDistribution() - Double-claim distribution by transferring vested governance tokens to new addresses

Description

DistributionProposal.registerForDistribution() snapshots dao.vestedBalance(msg.sender) into DistributionRedemption.registeredBalance[msg.sender], but claims/registration are only restricted per address, so the same vested governance tokens can be used to register multiple addresses and claim multiple times.

Vulnerability Details

During an active distribution, any user can call registerForDistribution() which records their current vested governance token balance as their claim entitlement. In DistributionRedemption, the only replay protection is registeredBalance[user] > 0 and hasClaimed[user], both keyed by address, not by tokens. Since MarketDAO.safeTransferFrom() allows transferring vested governance tokens (require(vestedBalance(from) >= amount)), a user can register with Address A, transfer the same vested governance tokens to Address B, register again, and later claim from both addresses, effectively duplicating entitlement.

```
uint256 vestedBal = dao.vestedBalance(msg.sender);

redemptionContract.registerClaimant(msg.sender, vestedBal);

if (registeredBalance[user] > 0) revert AlreadyRegistered();

registeredBalance[user] = governanceTokenBalance;

if (id == GOVERNANCE_TOKEN_ID && from != address(0)) {

require(vestedBalance(from) >= amount, "Cannot transfer unvested tokens");

}
```

Impact

An attacker can inflate total registered balances and drain the redemption pool by claiming multiple times with the same underlying governance tokens, potentially causing legitimate claimants to revert with InsufficientBalance and resulting in unfair/incorrect distributions.

Recommendation

Bind distribution to a non-transferable snapshot (e.g., snapshot balances at electionStart / a fixed snapshot block and compute claims from that snapshot), or prevent governance token transfers after distribution registration/snapshot, or require users to "lock/burn/escrow" governance tokens when registering so the same tokens cannot be used to register/claim across multiple addresses.

Status

Resolved

[H-03] Proposal#addSupport() - Proposal support can be double-counted by transferring vested governance tokens to new addresses

Description

Proposal.addSupport() accounts support per address and only checks the caller's current vested balance, so the same vested governance tokens can be used to add support multiple times by moving them across wallets.

Vulnerability Details

The addSupport(uint256 amount) function verifies dao.vestedBalance(msg.sender) and then increases both support[msg.sender] and supportTotal, but it does not lock tokens, track which tokens were used, or prevent the sender from transferring the same governance tokens after supporting. Because MarketDAO.safeTransferFrom() permits transferring vested governance tokens, a user can call addSupport() from Address A, transfer those vested tokens to Address B, and call addSupport() again from B, inflating supportTotal with the same underlying tokens and potentially triggering an election. .

```
uint256 availableBalance = dao.vestedBalance(msg.sender);

require(availableBalance >= amount, "Insufficient vested governance tokens");

require(support[msg.sender] + amount <= availableBalance, "Cannot support more than vested governance tokens held");

support[msg.sender] += amount;

supportTotal += amount;
```

Impact

An attacker can artificially increase supportTotal by reusing the same vested governance tokens across multiple addresses, which can wrongfully meet the support threshold and trigger elections for proposals without genuine backing, undermining governance integrity..

Recommendation

Use a snapshot-based mechanism for support (e.g., snapshot balances at proposal creation and cap per holder identity for that snapshot), or require staking/escrow of governance tokens when adding support (locking them until election trigger/expiration).

Status

Resolved

[H-04] Proposal#claimVotingTokens() - Voting power is not truly snapshotted at electionStart, enabling vote inflation via token transfers

Description

claimVotingTokens() is intended to mint voting tokens based on governance token holdings at electionStart, but it relies on vestedBalanceAt() which subtracts "locked-at-start" from the holder's current balance, allowing the same governance tokens to be transferred across fresh addresses during the election and used to mint voting tokens multiple times.

Vulnerability Details

During the election window, each address can call claimVotingTokens() once and it mints voting tokens equal to dao.vestedBalanceAt(msg.sender, electionStart). However, vestedBalanceAt() is not a true snapshot because it uses the holder's current governance token balance (balanceOf(holder, 0)) and only subtracts vesting schedules that were locked at electionStart. If the holder has no corresponding locked schedules, receiving vested governance tokens after electionStart increases vestedBalanceAt(..., electionStart) even though those tokens were not held at the snapshot time. Since MarketDAO.safeTransferFrom() allows transferring vested governance tokens, an attacker can claim voting tokens from Address A, transfer the same governance tokens to Address B during the election, and have B claim again, repeatedly minting extra voting tokens from the same underlying governance tokens.

Impact

An attacker can inflate voting power during an active election by reusing the same vested governance tokens across multiple addresses to mint voting tokens multiple times, potentially passing proposals without legitimate quorum/majority and breaking the integrity of the DAO's governance process.

Recommendation

Implement a real snapshot of governance token balances at electionStart (e.g., store per-address snapshot balances at election trigger, or use a snapshot-enabled token

design), and/or require governance tokens to be locked/escrowed when claiming voting tokens so they cannot be transferred and reused during the election window.

Status

Resolved

[H-05] MarketDao#safeTransferFrom() - Approved-operator can cast votes after election deadline

Description

MarketDAO.safeTransferFrom / safeBatchTransferFrom only blocks late vote transfers when msg.sender == from, so an approved ERC1155 operator can still transfer voting tokens to yesVoteAddress / noVoteAddress after the election ends, changing the vote totals used by execute() / failProposal().

Vulnerability Details

Votes are represented by ERC1155 voting tokens, and the protocol "counts" votes by reading the current balances held at the proposal's vote addresses. In MarketDAO.safeTransferFrom / safeBatchTransferFrom the "election still active" check only runs when msg.sender == from, e.g. if (_isActiveVotingToken(id) && msg.sender == from) { ... revert("Election has ended"); }. If a user has granted operator approval via setApprovalForAll, the operator can call safeTransferFrom(from, yesVoteAddress/noVoteAddress, votingTokenId, amount, ...) even after block.number >= electionStart + electionDuration, because msg.sender != from skips the check and the transfer proceeds. Proposal.execute() and failProposal() then use live balances to decide outcomes, e.g. uint256 yesVotes = dao.balanceOf(yesVoteAddress, votingTokenId); uint256 noVotes = dao.balanceOf(noVoteAddress, votingTokenId);, so post-deadline operator transfers directly modify yesVotes/noVotes right before finalization.

Impact

Election finality is broken: an approved operator can add votes after the deadline and potentially flip pass/fail outcomes (or meet/miss quorum) immediately before execute() / failProposal() is called.

Recommendation

Enforce the same "election must be active" restriction for vote-address transfers regardless of whether msg.sender == from (i.e., apply the check for any transfer to a registered vote address for a votingTokenId), or alternatively change finalization to use

an immutable snapshot of vote totals at election end so post-deadline transfers cannot affect results.

Status

Resolved

Medium

[M-01] DistributionProposal#initialize - Distribution payout pool is fixed at init but claims are registered using later vested balances

Description

The distribution proposal calculates and locks a fixed total payout amount when the proposal is initialized, but users register their claim using their current vested governance balance after the election starts, which may be higher. This can make total registered claims exceed the funded pool.

Vulnerability Details

DistributionProposal.initialize, the required funding is computed once using dao.getTotalVestedSupply() and stored as totalAmount. Later, after electionTriggered, users call registerForDistribution() which registers dao.vestedBalance(msg.sender) into the redemption contract. Since vestedBalance() can increase over time as vesting unlocks, the sum of all registeredBalance[user] * amountPerGovernanceToken can become larger than totalAmount. When claiming, DistributionRedemption.claim() computes claimAmount = registeredBalance[msg.sender] * amountPerGovernanceToken and reverts if the contract balance is not enough.

```
uint256 availableBalance = dao.vestedBalance(msg.sender);

require(availableBalance >= amount, "Insufficient vested governance tokens");

require(support[msg.sender] + amount <= availableBalance, "Cannot support more than vested governance tokens held");

support[msg.sender] += amount;

supportTotal += amount;
```

Impact

If the total registered entitlements become larger than the funded payout pool, the redemption contract can run out of funds and some users (often later claimers) will be unable to claim, causing failed redemptions and an unfair distribution outcome.

Recommendation

Use a consistent snapshot for sizing the pool and computing each user's registered entitlement, e.g., register using vestedBalanceAt(msg.sender, electionStart) (or another fixed snapshot block used for the distribution), or redesign payout calculation to be pro-rata against the actually funded pool so the contract cannot become underfunded.

Status

Resolved

[M-02] MarketDao#claimVestedTokens - Quorum/support calculations can use stale vested supply until vesting cleanup is claimed

Description

getTotalVestedSupply() is derived from a global counter of unvested tokens, but that counter is only updated when users clean up expired vesting schedules. If users do not call claimVestedTokens(), the vested supply used for support and quorum calculations can be lower than the effective vested balances.

Vulnerability Details

When users purchase tokens with vesting, totalUnvestedGovernanceTokens is increased. After vesting expires, the user's vestedBalance() will reflect the newly unlocked tokens, but totalUnvestedGovernanceTokens is not reduced unless _cleanupExpiredSchedules() is triggered via claimVestedTokens(). Since getTotalVestedSupply() returns totalSupply - totalUnvestedGovernanceTokens, it can remain understated until users perform cleanup. Governance then uses getTotalVestedSupply() to compute the support threshold and to snapshot snapshotTotalVotes at election start, so these values can be based on a stale vested-supply value.

```
function claimVestedTokens() external {

    require(hasClaimableVesting(msg.sender), "No vested tokens to claim");

    _cleanupExpiredSchedules(msg.sender);

}

function getTotalVestedSupply() public view returns (uint256) {

    uint256 total = tokenSupply[GOVERNANCE_TOKEN_ID];

    return total - totalUnvestedGovernanceTokens;

}

uint256 threshold = (dao.getTotalVestedSupply() * dao.supportThreshold()) / 10000;
```

Impact

Support thresholds and quorum snapshots may be calculated with a lower vested-supply value than expected, which can make it easier to trigger elections or satisfy quorum compared to a fully up-to-date vested supply.

Recommendation

Consider updating the unvested counter automatically when vesting expires (or add a permissionless function to clean up expired schedules for any holder), or base quorum/support calculations on a snapshot method that does not depend on manual vesting cleanup.

Status

Acknowledged; Document as known limitation rather than implement code fix.

[M-03] Escrow#_addDeposit() - Multiple Depositors Funds Ignored After First Deposit

Description

The escrow supports multiple deposits but only tracks and pays the first depositor, causing all subsequent depositors' funds to be permanently ignored.

Vulnerability Details

The `_addDeposit` function records each deposit, including the depositor's address, into the depositedAssets array. However, on the first deposit only, it sets `firstDepositor = depositor` and initializes the escrow lifecycle. The payment and settlement logic later relies solely on `firstDepositor` and does not iterate over or otherwise account for additional entries in `depositedAssets`.

```
function _addDeposit(
        AssetType assetType,
        address token,
        uint256 tokenId,
        uint256 amount,
        address depositor
    ) internal {
        // Start timer on first deposit
        if (depositedAssets.length == 0) {
            expiryTime = block.timestamp + duration;
            firstDepositor = depositor;
        }
        // Record the deposit
        depositedAssets.push(DepositedAsset({
            assetType: assetType,
            tokenAddress: token,
```

```
            tokenId: tokenId,
            amount: amount,
            depositor: depositor
        }));
    }
```

As a result, while multiple users can deposit assets and their deposits are stored in state, only the first depositor is ever considered as the payment recipient. All subsequent depositors have no mechanism to receive funds back or be included in settlement, even though their assets are locked in the escrow.

Impact

All depositors except the first permanently lose access to their funds, leading to direct financial loss and broken escrow guarantees in multi-user scenarios.

Recommendation

We recommend modifying the `_addDeposit` function and settlement logic to either explicitly restrict deposits to a single depositor or correctly track and settle funds for all depositors instead of relying on `firstDepositor`.

Status

Resolved

Low

[L-01] Escrow#_executeSwap() - Native Payment Can Be Permanently Blocked By Recipient

Description

The swap finalization performs an immediate native token transfer to the first depositor, which can fail if the recipient contract cannot accept ETH, blocking the entire swap flow.

Vulnerability Details

The `_executeSwap` function finalizes the escrow by transferring the payment to firstDepositor before distributing deposited assets to the buyer. When `paymentAssetType` is `AssetType.NATIVE`, the function uses a low-level `.call{value: paymentAmount}("")` to transfer ETH.

If `firstDepositor` is a smart contract without a `receive()` or `fallback()` function, or one that intentionally reverts on ETH reception, the call fails and the function reverts with `TransferFailed()`. Since `completed` is set before the transfer attempt, the escrow becomes permanently stuck: the buyer cannot receive the assets, and no retry or alternative payment path exists. The root cause is the push-based native payment design combined with an assumption that the recipient can always accept ETH.

Impact

The escrow can be permanently denied settlement, preventing buyers from receiving assets and locking funds in the contract, resulting in loss of usability and potential financial damage.

Recommendation

We recommend modifying the `_executeSwap` function to use a pull-based claim mechanism for payments, allowing depositors to claim their native funds later instead of requiring an immediate ETH transfer during swap execution.

Status

Acknowledged; Won't Fix - push-based settlement is intentional design.

[L-02] Escrow#_executeSwap() - Malicious Depositor Can Grief Native Payment Transfer

Description

Native token transfers rely on a low-level .call, allowing a malicious recipient contract to intentionally consume excessive gas and force the swap to revert.

Vulnerability Details

In `_executeSwap`, when `paymentAssetType` is `AssetType.NATIVE`, the contract transfers ETH to firstDepositor using `(bool success, ) = firstDepositor.call{value: paymentAmount}("")`.

If `firstDepositor` is a malicious contract, its `fallback` or `receive` function can deliberately execute gas-intensive logic or revert. This causes the `.call` to fail, triggering `TransferFailed()` and reverting the entire swap. Because the payment transfer is mandatory for swap completion, the attacker can repeatedly grief settlement without needing to steal funds. The root cause is a push-based ETH transfer that assumes cooperative recipient behavior.

Impact

A malicious first depositor can permanently block swap execution, preventing buyers from receiving assets and causing a denial of service for the escrow.

Recommendation

We recommend updating the `_executeSwap` function to replace direct native token transfers with a pull-based claim mechanism, allowing depositors to withdraw their ETH independently and preventing griefing via fallback gas abuse.

Status

Acknowledged; Won't Fix - push-based settlement is intentional design.

[L-03] Escrow#_transferAsset() - Unsafe ERC20 Transfer Usage

Description

The asset transfer logic uses `ERC20.transfer`, which is unsafe for non-standard ERC20 tokens and may cause silent failures or unexpected reverts.

Vulnerability Details

In `_transferAsset`, ERC20 assets are transferred using `IERC20(token).transfer(recipient, amount)` and the return value is checked. However, many widely used ERC20 tokens do not strictly adhere to the ERC20 standard: some do not return a boolean value, while others revert on failure instead of returning `false`.

Using `transfer` directly makes the function incompatible with such tokens and can lead to unexpected reverts or failed transfers, breaking asset delivery. The root cause is reliance on raw ERC20 transfers instead of a safe abstraction that handles these edge cases.

Impact

ERC20 asset transfers may fail unexpectedly or behave inconsistently, potentially preventing buyers from receiving assets and causing denial of service for the escrow flow.

Recommendation

We recommend updating the `_transferAsset` function to use `safeTransfer` from a well-audited library (e.g., OpenZeppelin `SafeERC20`) to ensure compatibility with both standard and non-standard ERC20 tokens.

Status

Resolved

[L-04] Escrow,MarketDao# - External-call reentrancy surface in fund-moving functions (Escrow swap transfers + MarketDAO proposal ETH transfers)

Description

Both Escrow and MarketDAO execute external calls while moving assets (ETH sends and token transfers) without a reentrancy guard, which increases reentrancy and control-flow risk during transfers.

Vulnerability Details

MarketDAO.transferETH forwards ETH using a low-level call to an arbitrary recipient, giving the recipient contract control during execution and enabling reentrant calls back into the DAO. Escrow performs multiple external interactions during swap execution (ETH/token transfers) that can also allow reentrant control-flow through untrusted recipients or token contracts, even though swap state is largely protected by the early completed flag.

Impact

Reentrancy during these external calls can cause unexpected behavior or denial of service (e.g., reentering and reverting transfers), and it becomes a direct fund-risk if surrounding logic later adds additional state changes or accounting around these calls.

Recommendation

Add a reentrancy guard and apply it to transferETH (and other transfer functions) and to the escrow's swap-triggering entrypoints as defense-in-depth, and keep checks-effects-interactions or use pull-based withdrawals where possible.

Status

Resolved

QA

[Q-01] Escrow - Missing Event Emission On Important Functions

Vulnerability Details

The contract performs key state transitions such as recording deposits, executing swaps, and transferring assets without emitting any events. Events are the primary mechanism for off-chain services, indexers, frontends, and auditors to track protocol activity.

Without events, users cannot reliably monitor deposits or settlements, and integrations must rely on costly and error-prone state polling. The root cause is the absence of event definitions and `emit` statements for major escrow lifecycle actions.

Recommendation

We recommend updating the Escrow contract to emit events for deposits, swap execution, and withdrawals to ensure proper tracking and off-chain observability.

Status

Resolved

[Q-02] Escrow - Use of floating pragma

Vulnerability Details

Multiple contracts use floating pragma version (^0.8.20). It's crucial to compile and deploy contracts with the same compiler version and settings used during development and testing. Fixing the pragma version prevents compilation with different versions. Using an outdated pragma version could introduce bugs that negatively affect the contract system, while newly released versions may have undiscovered security vulnerabilities.

Recommendation

Change the pragma statement in contracts to a fixed version, such as pragma solidity 0.8.20;. This locks the contract to a specific compiler version.

Status

Acknowledged; Keep floating pragma for OpenZeppelin compatibility.

[Q-03] Escrow - Missing input validation in Escrow.initialize() can allow invalid escrow configurations

Vulnerability Details

Escrow contracts sets critical parameters in initialize() without validating them, for example duration = _duration and the payment asset configuration fields.

Recommendation

Add basic parameter checks in Escrow.initialize() such as require(_duration > 0), validate (assetType, token, tokenId, amount) combinations, and reject zero-amount payments for NATIVE/ERC20/ERC1155 unless explicitly intended.

Status

Resolved

[Q-04] Proposal - Proposal description validation is inconsistent across proposal types

Vulnerability Details

ResolutionProposal and ParameterProposal require a non-empty description, but TreasuryProposal, MintProposal, and DistributionProposal do not enforce this and accept an empty string. This can lead to proposals being created without meaningful metadata, which reduces clarity for reviewers and voters.

Recommendation

Apply a consistent rule across all proposal types, such as requiring bytes(_description).length > 0 in every proposal initialize() (or enforce it centrally in the factory if you prefer one place).

Status

Acknowledged; Some proposal types require non-empty description, others don't.

Centralisation

The MarketDAO project values decentralisation alongside security and utility.

The protocol's functions are designed to operate independently, minimising reliance on the internal team and hardcoded values, while maintaining robust security and functionality.

Conclusion

After Hashlock's analysis, the MarketDAO 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