shieldify

Pudgy Daos

SECURITY REVIEW

Date: 11 June 2025

CONTENTS

1. **About Shieldify Security** 3
2. **Disclaimer** 3
3. **About Pudgy Daos** 3
4. **Risk classification** 3
   - 4.1 Impact 3
   - 4.2 Likelihood 3
5. **Security Review Summary** 4
   - 5.1 Protocol Summary 4
   - 5.2 Scope 4
6. **Findings Summary** 4
7. **Findings** 5

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Pudgy Daos

Pudgy Daos is a platform built on the Abstract blockchain that empowers users to easily create, fund, and manage Decentralized Autonomous Organizations (DAOs). It lowers the technical and financial barriers for creators and communities to launch their own DAO projects, enabling decentralized governance and capital formation.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 6 days, with a total of 96 hours dedicated by 2 researchers from the Shieldify team.

Overall, the code is well-written. The audit report flagged six High and four Medium severity issues, mainly related to the possibility of draining vested tokens through the claiming functionality, the missing withdrawal mechanism, lack of a deadline parameter, predictable token deployment address, and a risk of tokens getting stuck due to the `finalize()` function.

Shieldify has audited the proposed fixes for the identified findings which the team have promptly implemented.

5.1 Protocol Summary

Project Name	**Pudgy Daos**
Repository	daoslive-sc
Type of Project	DAOs
Audit Timeline	6 days
Review Commit Hash	9a1856db2060b609a17b24aa72ab35f2cdf09031
Fixes Review Commit Hash	f65c39d825a0e66f4f5476f9ce72635f50ff4cff

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/DaosLocker.sol	82
contracts/DaosToken.sol	28
contracts/DaosVesting.sol	126
contracts/DaosLive.sol	291
contracts/DaosFactory.sol	153
Total	**680**

6. Findings Summary

The following issues have been identified, sorted by their severity:

- **Critical/High** issues: **6**
- **Medium** issues: **4**
- **Low** issues: **7**
- **Info** issues: **4**

ID	Title	Severity	Status
[C-01]	Complete Drainage of Vested Tokens from `claim()` in `DaoVesting`	Critical	Fixed
[C-02]	The `deadline` Parameter in UniswapV3 Swap Causes Sniping Functionality Failure in `finalize()` Function	Critical	Fixed
[H-01]	Swapped Tokens During `finalize()` Permanently Stuck in `DaosLive` Due to Missing Withdrawal Mechanism	High	Fixed
[H-02]	In `finalize()`, the `_mintToDao` Amount of Tokens Are Permanently Stuck in `DaosLive`	High	Fixed
[H-03]	Predictable Token Deployment Address Enables Liquidity Pool Exploit in `finalize()` in `DaosLive`	High	Fixed
[H-04]	Fee Theft via Arbitrary Contract Impersonation in `collect()` Function in `DaosLocker`	High	Fixed
[M-01]	`DaoFactory` Inherits `PausableUpgradeable` but Does Not Implement any Pausable Functionalities	Medium	Fixed
[M-02]	Refund Block via `extendTime()` Function in `DaosLive`	Medium	Fixed
[M-03]	The `finalize()` Function DoS if Vesting Gets Scheduled Before Its Execution	Medium	Fixed
[M-04]	Premature `setGoalReached()` Call Disrupts Token Proportions and Refunds	Medium	Fixed
[L-01]	Potential Precision Loss Due to Imbalanced Schedule Configuration	Low	Fixed
[L-02]	Excessively Broad Tick Range Dilutes Liquidity on Uniswap V3	Low	Acknowledged
[L-03]	Excess Fee Not Refunded in `DaosVesting`	Low	Fixed
[L-04]	Incorrect Funding Phase Validation in `DaosLive`	Low	Fixed
[L-05]	Vesting Schedule Can Be Manipulated by DAO Manager	Low	Fixed
[L-06]	Unrestricted Vesting End Date in Schedule Setting	Low	Fixed
[L-07]	Unfair Token Distribution Due to Discount Mechanism	Low	Fixed
[I-01]	Redundant NFT Receiver Functions in `DaosLive` Contract	Info	Fixed
[I-02]	Missing Proper Time Validation in `create()` Function in `DaosFactory`	Info	Fixed
[I-03]	Redundant `receive()` Function in `DaosLive`	Info	Fixed
[I-04]	Missing Strict Checks for Vesting Start Time Validation	Info	Fixed

7. Findings

[C-01] Complete Drainage of Vested Tokens from `claim()` in `DaoVesting`

Severity

Critical Risk

Description

The `DaosVesting::claim()` function has a critical vulnerability that allows an attacker to drain tokens from legitimate DAOs. The issue stems from insufficient validation of the DAO contract address and its state.

Vulnerable code in `DaosVesting::claim()`:

```
function claim(address dao, address user, uint256 index) external {
    IDaosLive daosLive = IDaosLive(dao);
@-> address token = daosLive.token();

@-> uint256 maxPercent = getClaimablePercent(dao, index);
    if (maxPercent > DENOMINATOR) revert InvalidCalculation();

    unchecked {
@->     uint256 totalAmount = daosLive.getContributionTokenAmount(user);
        uint256 maxAmount = (totalAmount * maxPercent) / DENOMINATOR;
        if (maxAmount < claimedAmounts[dao][user]) {
            revert InvalidCalculation();
        }
        uint256 amount = maxAmount - claimedAmounts[dao][user];

        if (amount > 0) {
            claimedAmounts[dao][user] += amount;
            totalClaimeds[dao] += amount;
@->          TransferHelper.safeTransfer(token, user, amount);
            observer.emitClaimed(dao, token, user, amount);
        }
    }
}
```

Attack Vector: 1. Attacker deploys fake malicious contract implementing `IDaosLive` 2. Sets `token` address to a legitimate DAO's token 3. Implements `getContributionTokenAmount()` to manually return the amount of legit `DaosLive` tokens locked in `DaoVesting.sol` contract. 4. Sets vesting schedules of that Fake `IDaosLive` implementation with 100% unlock for an `index` . 5. Calls `claim()` on the vesting contract with their Fake malicious `DoasLive` contract address 6. Receives tokens from the legitimate DAO's vesting contract

Location of Affected Code

File: contracts/DaosVesting.sol#L91

```
function claim(address dao, address user, uint256 index) external {
    IDaosLive daosLive = IDaosLive(dao);
@-> address token = daosLive.token();

@-> uint256 maxPercent = getClaimablePercent(dao, index);
    if (maxPercent > DENOMINATOR) revert InvalidCalculation();

    unchecked {
@->     uint256 totalAmount = daosLive.getContributionTokenAmount(user);
        uint256 maxAmount = (totalAmount * maxPercent) / DENOMINATOR;
        if (maxAmount < claimedAmounts[dao][user]) {
            revert InvalidCalculation();
        }
        uint256 amount = maxAmount - claimedAmounts[dao][user];

        if (amount > 0) {
            claimedAmounts[dao][user] += amount;
            totalClaimeds[dao] += amount;
@->          TransferHelper.safeTransfer(token, user, amount);
            observer.emitClaimed(dao, token, user, amount);
        }
    }
}
```

Impact

- Complete drain of tokens from legitimate DAOs
- Loss of all funds of vested tokens that should have been claimed by the real contributors of `Pudgy Daos`
- Permanent loss of tokens
- Affects all DAOs using the vesting contract.

Recommendation

- Restrict the caller to claim the tokens only through `DaosLive` contract function, claim only. Do not allow users to clam directly through the vesting contract.
- Validate that the DAO contract is legitimate or not.

Team Response

Fixed.

[C-02] The `deadline` Parameter in UniswapV3 Swap Causes Sniping Functionality Failure in `finalize()` Function

Severity

Critical Risk

Description

In the following code snippet, using `IUniRouter.exactInputSingle()` for token swapping:

```
IUniRouter uniRouter = IUniRouter(_factory.uniRouter());
uniRouter.exactInputSingle{value: snipeAmount}(
    IUniRouter.ExactInputSingleParams({
        tokenIn: wethAddr,
        tokenOut: token,
        fee: UNISWAP_V3_FEE,
        recipient: address(this),
        amountIn: snipeAmount,
        amountOutMinimum: 0,
        sqrtPriceLimitX96: 0
    })
);
```

The `deadline` parameter is missing during the initialization of `ExactInputSingleParams`. As a result, the default value of `0` is used for `deadline`, which is always in the past. This causes the Uniswap `exactInputSingle()` function to **revert** the transaction due to deadline expiry in the below Uniswapv3 code.

SwapRouter.sol

```
function exactInputSingle(ExactInputSingleParams calldata params)
    external
    payable
    override
@>  checkDeadline(params.deadline)
    returns (uint256 amountOut)
{
    amountOut = exactInputInternal(
        params.amountIn,
        params.recipient,
        params.sqrtPriceLimitX96,
        SwapCallbackData({path: abi.encodePacked(params.tokenIn, params.
            fee, params.tokenOut), payer: msg.sender})
    );
    require(amountOut >= params.amountOutMinimum, 'Too little received');
}
```

PeripheryValidation.sol

```
abstract contract PeripheryValidation is BlockTimestamp {
    modifier checkDeadline(uint256 deadline) {
        require(_blockTimestamp() <= deadline, 'Transaction too old');
        _;
    }
}
```

Location of Affected Code

File: contracts/DaosLive.sol#L314

Impact

- All token snipe attempts will fail and revert unless `deadline` is correctly set.
- Results in denial of service for token snipe functionality, along with failure of `finalize()` function.

Recommendation

Set the `deadline` parameter explicitly using a valid timestamp (e.g., `block.timestamp` at least the current timestamp).

```
exactInputSingleParams({
    tokenIn: wethAddr,
    tokenOut: token,
    fee: UNISWAP_V3_FEE,
    recipient: address(this),
    amountIn: snipeAmount,
    amountOutMinimum: 0,
    sqrtPriceLimitX96: 0,
+   deadline: block.timestamp
})
```

Team Response

Fixed.

[H-01] Swapped Tokens During `finalize()` Permanently Stuck in `DaosLive` Due to Missing Withdrawal Mechanism

Severity

High Risk

Description

When performing a token swap for `snipeAmount` in the `finalize()` function, using the `IUniRouter.exactInputSingle()` function in the following code snippet:

```
IUniRouter uniRouter = IUniRouter(_factory.uniRouter());
uniRouter.exactInputSingle{value: snipeAmount}(
    IUniRouter.ExactInputSingleParams({
        tokenIn: wethAddr,
        tokenOut: token,
        fee: UNISWAP_V3_FEE,
        recipient: address(this), // swapped tokens sent here
        amountIn: snipeAmount,
        amountOutMinimum: 0,
        sqrtPriceLimitX96: 0
    })
);
```

The `recipient` of the swapped tokens is set to `address(this)`, i.e., the `DaosLive.sol` contract. However, the contract lacks any function or mechanism to retrieve or forward these tokens, resulting in the swapped tokens being permanently stuck in the contract.

Location of Affected Code

File: contracts/DaosLive.sol#L314

Impact

Swapped tokens are locked and inaccessible forever.

Recommendation

- Implement a **withdrawal** or **sweep** function in `DaosLive.sol` that allows an authorized address (e.g., owner or governance) to recover any ERC20 tokens held by the contract.
- Alternatively, consider making the `recipient` parameter configurable or passed in explicitly, allowing swapped tokens to be sent to a retrievable address.

Team Response

Fixed.

[H-02] In `finalize()`, the `_mintToDao` Amount of Tokens Are Permanently Stuck in `DaosLive`

Severity

High Risk

Description

In the `DaosLive::finalize()` function, there's a parameter `_mintToDao()` that allows minting additional tokens to the contract address.

These tokens are minted to the contract address (`address(this)`), but there is no functionality implemented to allow the DAO manager to access or utilize these tokens. The tokens become permanently locked in the contract.

Location of Affected Code

File: contracts/DaosLive.sol

```
 function finalize(
    string calldata name,
    string calldata symbol,
    IDaosVesting.Schedule[] calldata schedules,
@-> uint256 _mintToDao,
    uint256 _snipeAmount // eth
) external onlyOwner {
// code

    uint256 mintToDao = _mintToDao;
    if (mintToDao > 0) {
@->     IDaosToken(token).mint(address(this), mintToDao);
    }

// code
}
```

Impact

- Tokens minted through `_mintToDao()` are permanently locked in the contract, leading to a permanent loss of tokens, as there's no way to recover them
- Affects all DAOs that use the `_mintToDao()` parameter during `finalize()`

Recommendation

- Add a dedicated function to transfer tokens to the DAO manager.
- Modify the `finalize()` function to mint directly to the DAO manager or any recipient address.

Team Response

Fixed.

[H-03] Predictable Token Deployment Address Enables Liquidity Pool Exploit in `finalize()` in `DaosLive`

Severity

High Risk

Description

An attacker can exploit the deterministic nature of contract deployment using the `CREATE` opcode to precompute the address of the `DaosToken` deployed within the `DaosLive::finalize()` function. This predictability enables a front-running attack that severely impacts the liquidity and valuation of the token in Uniswap v3 pools.

The `DaosToken` is deployed via the `CREATE` opcode inside the `finalize()` function of the `DaosLive` contract. The use of `CREATE` makes the address of the deployed contract predictable using the sender address and nonce:

> **Source:** evm.codes
> The destination address is calculated as the rightmost 20 bytes (160 bits) of the Keccak-256 hash of the RLP encoding of the sender address, followed by its nonce.
> `address = keccak256(rlp([sender_address, sender_nonce]))[12:]`

By leveraging this predictability, an attacker can execute the following exploit sequence:

Exploit Scenario

1. The attacker precomputes the future address of the `DaosToken` contract as it is deployed via the `CREATE` opcode using this: `keccak256(rlp([deployer_address, deployer_nonce]))[12:]` .
2. The attacker creates a Uniswap v3 liquidity pool between WETH and the precomputed `DaosToken` address using the same fee tier and tick range expected in the `finalize()` function.
3. The attacker adds liquidity to the newly created pool by supplying a very small amount of WETH and a large amount of spoofed `DaosToken` .
4. The spoofed token is accepted by Uniswap because the address exists even though the actual contract code is not yet deployed.
5. The attacker waits for the `DaosLive::finalize()` function to be executed by the protocol.
6. Upon execution, the `DaosToken` contract is deployed at the precomputed address, and the protocol adds a significant amount of real `DaosToken` and WETH as liquidity to the same pool.
7. The attacker removes their liquidity position from the pool, receiving a large share of the real `DaosToken` just added by the protocol.
8. The attacker then swaps all the extracted `DaosToken` for WETH in the same pool.
9. The swap operation drains a significant portion of WETH from the pool, resulting in a skewed token price.
10. The final state of the pool has almost no WETH and an imbalanced supply of `DaosToken` , severely devaluing the token and harming the protocol's liquidity.

Location of Affected Code

File: contracts/DaosLive.sol

```
function finalize(
// code
  bytes memory data = abi.encodeWithSelector(
      IDaosToken.initialize.selector,
      address(_factory),
      name,
      symbol
  );
  BeaconProxy tokenBeacon = new BeaconProxy(_factory.tokenBeacon(), data)
     ;
  token = address(tokenBeacon);
  _factory.emitTokenCreated(token);

// code
}
```

Impact

- **Loss of Protocol Funds:** Protocol-owned WETH is extracted via manipulated liquidity removal.
- **Price Collapse:** The `DaosToken` becomes nearly worthless on Uniswap due to the attacker draining liquidity.

Recommendation

To mitigate this issue:

Use `CREATE2` with a salt:

- This allows the `Pudgy Daos` owner to control and obscure the final address more securely.

Team Response

Fixed.

[H-04] Fee Theft via Arbitrary Contract Impersonation in `collect()` Function in `DaosLocker`

Severity

High Risk

Description

The `DaosLocker::collect()` function is vulnerable to impersonation attacks. A malicious actor can deploy an **arbitrary contract** that mimics the **token address and LP token ID** of a legitimate `DaosLive` contract. By doing so, they can illegitimately invoke `collect()` and **claim all the accrued swap fees**, effectively **stealing earnings meant for the rightful owner**.

Exploit Scenario

1. A malicious actor observes the finalized `DaosLive` contract.
2. They deploy a **fake contract** that uses the **same token address and LP token ID** as the original `DaosLive` contract.
3. The attacker then calls `DaosLocker::collect()` using the malicious contract.
4. The function executes and **transfers the Dao fees** to the attacker — fees that were originally intended for the legitimate owner of the `DaosLive` contract.

Root Cause

- Lack of **authorization checks** on the caller of `collect()`
- Absence of **contract ownership or identity verification**

Location of Affected Code

File: contracts/DaosLocker.sol

```
function collect(address dao) external {
// code
  IDaosLive daosLive = IDaosLive(dao);
  @>  address token = daosLive.token();
  INonfungiblePositionManager positionManager =
     INonfungiblePositionManager(
        _factory.uniV3PositionManager()
  );
  address wethAddr = positionManager.WETH9();

  (uint256 amount0, uint256 amount1) = positionManager.collect(
      INonfungiblePositionManager.CollectParams({
          recipient: address(this),
          amount0Max: type(uint128).max,
          amount1Max: type(uint128).max,
   @>     tokenId: daosLive.lpTokenId()
      })
  );

  TransferHelper.safeTransfer(
        wethAddr,
  @>    OwnableUpgradeable(dao).owner(),
            daoEth
        );
        TransferHelper.safeTransfer(
            token,
        @>  OwnableUpgradeable(dao).owner(),
            daoToken
        );
// code
}
```

Impact

Drain all accumulated Uniswap swap fees that were meant for legitimate DAOs. **Completely bypass the intended ownership model**, allowing external actors to claim protocol rewards.

Recommendation

Consider implementing the steps described below to mitigate this issue: - Create a function inside `DaosLive` to call `DaosLocker::collect()` - Instead of taking input, use `msg.sender` for Dao address

Team Response

Fixed.

[M-01] `DaoFactory` Inherits `PausableUpgradeable` but Does Not Implement any Pausable Functionalities

Severity

Medium Risk

Description

The `DaosFactory` contract inherits from `PausableUpgradeable` but fails to implement the pause functionality in the `create()` function.

The function is missing the `whenNotPaused` modifier that should be inherited from `PausableUpgradeable` .

Location of Affected Code

File: contracts/DaosFactory.sol#L88

```
function create(
    uint256 supply,
    uint256 totalSale,
    uint256 fundraisingGoal,
    uint256 endTime
@-> ) external returns (address instance) {
    if (fundraisingGoal == 0) revert InvalidFundraisingGoal();
    if (totalSale == 0) revert InvalidTotalSale();
    if (totalSale > (supply * 9) / 10) revert TotalSaleTooLarge();
    if (totalSale < supply / 2) revert TotalSaleTooSmall();
    if (endTime < block.timestamp) revert InvalidEndTime();
// code
}
```

Impact

Affects the entire protocol's ability to respond to emergencies. Due to any emergency reason, if the protocol owner wants to pause the creation of new `DaosLive.sol` instances from being created, then they cannot do it.

Recommendation

Add the `whenNotPaused` modifier to the `create()` function:

```
function create(
    uint256 supply,
    uint256 totalSale,
    uint256 fundraisingGoal,
    uint256 endTime
-   ) external returns (address instance) {
+   ) external whenNotPaused returns (address instance) {
    if (fundraisingGoal == 0) revert InvalidFundraisingGoal();
    if (totalSale == 0) revert InvalidTotalSale();
    if (totalSale > (supply * 9) / 10) revert TotalSaleTooLarge();
    if (totalSale < supply / 2) revert TotalSaleTooSmall();
    if (endTime < block.timestamp) revert InvalidEndTime();
    // code
}
```

Team Response

Fixed.

[M-02] Refund Block via `extendTime()` Function in `DaosLive`

Severity

Medium Risk

Description

The `DaosLive` contract allows the owner to **arbitrarily block refunds** by repeatedly extending the `endTime` of the DAO using the `extendTime()` function. Since refunds are only available after `endTime`, this introduces a **centralization risk** and **griefing vector** for contributors.

Attack Scenario

1. Users contribute to the DAO expecting to be able to claim refunds if the project does not finalize.
2. The `endTime` parameter determines when refunds can be claimed.
3. The contract allows the owner to call `extendTime()` and increase `endTime`.
4. The owner can call `extendTime()` repeatedly before each expiration.
5. This results in a loop where `endTime` is always in the future, **blocking refunds forever**.

Location of Affected Code

File: contracts/DaosLive.sol#L333

```
function extendTime(uint256 _endTime) external onlyOwner {
    if (goalReached) revert GoalReached();
    if (_endTime < block.timestamp) revert InvalidEndTime();
    endTime = _endTime;
    _factory.emitExtendTime(endTime);
}
```

Impact

- Refunds can be indefinitely delayed, effectively trapping user funds and denying contributors their right to exit.
- The contract introduces a significant centralization risk, as the owner can unilaterally manipulate the refund timeline.
- This creates a potential griefing or malicious vector, where the owner may extend the `endTime` continuously to prevent withdrawals or even by setting it to a very high value that is not possible to reach in the near future.

Recommendation

- Impose a strict cap on the total amount of time that `endTime` can be extended.
- Require governance approval (e.g., DAO vote or multisig) for any extension beyond a predefined threshold.

Team Response

Fixed.

[M-03] The `finalize()` Function DoS if Vesting Gets Scheduled Before Its Execution

Severity

Medium Risk

Description

In `DaosLive.sol` contract, there is a function `setSchedules()` that allows the `daoManager` to set the vesting schedule (Here this function was supposed to be called by `daoManager` ONLY when they want to make a new vesting schedule and delete the old vesting schedule) However, here a problem arrises as function `setSchedules()` does NOT hold any check that allows the `daoManager` to call this function ONLY after `finalize()` function is called. Meaning this function can be called even before the function `finalize()` to set the vesting schedule.

```
function setSchedules(IDaosVesting.Schedule[] calldata schedules)
    external payable onlyOwner {
     _factory.vesting().setSchedules{value: msg.value}(schedules);
}
```

If the above scenario that we discussed took place, then due to that, the entire `finalize()` function will be DoS permanently, because now, as the vesting schedule is already set `schedules[msg.sender].length > 0` true for this `DaosLive` contract, it will need the `0.5 ether` override fees to set the new schedule.

Here `DaosVesting.sol`

```
 function setSchedules(Schedule[] calldata _schedules) external payable {
// only if he has already build schedule.
@->     if (schedules[msg.sender].length > 0) {
// to overide the schedule of vesting the owner needs to pay 0.5 ether.
@->         if (msg.value < overrideFee) revert InvalidFee();
            delete schedules[msg.sender];
        }
}
```

And as the function `finalize()` calls `_factory.vesting().setSchedules(schedules);` here we do not pass any ether `override` fees, also the function `finalize()` is itself NOT `payable`

```
function finalize(
    string calldata name,
    string calldata symbol,
    IDaosVesting.Schedule[] calldata schedules,
    uint256 _mintToDao,
    uint256 _snipeAmount
) external onlyOwner {
// code
@-> _factory.vesting().setSchedules(schedules);  // No value sent here
// code
}
```

Location of Affected Code

File: contracts/DaosVesting.sol#L46

File: contracts/DaosLive.sol#L227

Impact

- Permanent DoS of the `finalize()` function
- Contributors' ETH permanently locked (can not call fn Refund)
- DAO Manager's funds locked (can not call fn execute)
- Protocol enters an irrecoverable state of DoS

Recommendation

Consider not allowing the owner to call `setSchedules()` before the `finalize()` function is called.

```
function setSchedules(IDaosVesting.Schedule[] calldata schedules)
    external payable onlyOwner {
+    if (token == address(0)) revert NotFinalized();
     _factory.vesting().setSchedules{value: msg.value}(schedules);
     }
}
```

Team Response

Fixed.

[M-04] Premature `setGoalReached()` Call Disrupts Token Proportions and Refunds

Severity

Medium Risk

Description

A critical issue arises in the `DaosLive.sol` contract where the owner can prematurely call the `setGoalReached()` function before the `fundraisingGoal` is actually met. This action has two severe consequences:

1. **Disruption of Agreed Token Proportions:**
   When the `fundraisingGoal` is not actually reached but the owner still calls the `setGoalReached()` function, it results in incorrect token minting. More tokens are minted relative to the actual liquidity raised, which disrupts the originally agreed-upon token proportions defined during the contract creation. This creates an imbalance and is unfair to contributors, as the token distribution no longer reflects the intended fundraising outcome.
2. **Refund Blockage for Users:**
   Once the `goalReached` flag is set to `true`, users can no longer claim refunds. Even if the fundraising goal was not genuinely achieved, users are unfairly locked out from retrieving their funds, Because they contributed based on the agreed values at the time of contract creation.

Attack Scenario

1. The contract is deployed with a specified `fundraisingGoal`.
2. The owner manually invokes the `setGoalReached()` function before the actual fundraising goal is met, regardless of any intention.
3. The system assumes the goal is legitimately reached:
   - Token proportions are finalized based on incorrect assumptions.
   - Refund mechanism is permanently disabled (as `goalReached == true`).
4. Contributors are misled and lose access to their funds, despite the goal being unmet.

Location of Affected Code

File: contracts/DaosLive.sol

```
function setGoalReached() external onlyOwner {
    goalReached = true;
    _factory.emitGoalReached(totalRaised);
}
```

Impact

- Inaccurate token distribution/management.
- Unfair locking of contributor funds, as they have no opportunity to refund even when the `fundraisingGoal` amount is not reached.

Recommendation

- Implement a validation check within the `setGoalReached()` function to ensure it can only be called if the actual `fundraisingGoal` has been met. For example:

```
require(totalRaised >= fundraisingGoal, "Fundraising goal not
   reached yet");
```

[OR]

- Have a minimum x% of `fundraisingGoal` to be at least completed as `totalRaised` before this function can be called, so that the token ratio will not be dropped heavily as compared to the token proportion promised.

Eg, at least 80% of the `fundraisingGoal` must be completed.

```
function setGoalReached() external onlyOwner {
    if (fundraisingGoal * 8000 / DENOMINATOR > totalRaised) revert();
        goalReached = true;
        _factory.emitGoalReached(totalRaised);
    }
}
```

Team Response

Fixed.

[L-01] Potential Precision Loss Due to Imbalanced Schedule Configuration

Severity

Low Risk

Description

When the total scheduled duration is significantly long (e.g., 1 month) and the `schedule.period` is relatively short (e.g., 1 hour), the computed `totalPeriods` can become disproportionately large. In such cases, if both `schedulePercent` and `completedPeriods` are small, the expression `schedulePercent * completedPeriods` may be smaller than `totalPeriods`, leading to incorrect logic execution. This misalignment can result in unintended behavior, especially in time-based unlocking, vesting, or reward mechanisms.

Location of Affected Code

File: contracts/DaosVesting.sol

```
function getClaimablePercent( address dao, uint256 index ) public view
    returns (uint256) {
// code
  uint256 schedulePercent = schedule.unlockPercent - basePercent;
// code
  return basePercent + (schedulePercent * completedPeriods) /
      totalPeriods;
}
```

Recommendation

Use fixed-point arithmetic with a precision multiplier (e.g., `1e18`) to preserve fractional values.

Team Response

Fixed.

[L-02] Excessively Broad Tick Range Dilutes Liquidity on Uniswap V3

Severity

Low Risk

Description

The `MIN_TICK` and `MAX_TICK` parameters are configured so broadly that the liquidity position spans almost the entire Uniswap V3 price curve. In effect, this causes the position to behave like a Uniswap V2 pool, providing liquidity across all prices, rather than concentrating it within a strategic range. The immediate consequences are:

- **Liquidity Dilution**: Capital is spread thinly across an enormous range, reducing depth at any given price point.
- **Lower Fee Capture**: Because most trades occur within a narrower real-world price band, only a small fraction of the position actively earns swap fees, resulting in diminished returns for the `DaosLive` contract
- **Capital Inefficiency**: Funds locked in price zones far from the market are idle, earning no meaningful fees while still bearing opportunity cost.

Location of Affected Code

File: contracts/DaosLive.sol

```
int24 internal constant MIN_TICK = -887272;
int24 internal constant MAX_TICK = -MIN_TICK;
```

Recommendation

Define a tighter tick range tailored to the expected trading band of the paired assets. Practical steps include: - **Periodic Rebalancing**: Implement an automated or governance-triggered mechanism to adjust the tick range as market conditions evolve, keeping liquidity concentrated where volume is highest.

Team Response

Acknowledged.

[L-03] Excess Fee Not Refunded in `DaosVesting`

Severity

Low Risk

Description

In `DaosVesting::setSchedules()`, if a user sends more ETH than the required `overrideFee`, that excess amount of ETH is lost forever. The function only checks if the sent amount is less than the fee (`msg.value < overrideFee`), but doesn't handle cases where more ETH is sent than necessary.

Location of Affected Code

File: contracts/DaosVesting.sol

Recommendation

Implement a refund mechanism to return any excess ETH sent above the required override fee to the sender. This will prevent funds from being unnecessarily lost and improve the user experience.

Team Response

Fixed.

[L-04] Incorrect Funding Phase Validation in `DaosLive`

Severity

Low Risk

Description

The `onlyFundingPhase()` modifier in `DaosLive` has a logical error in its condition. The current implementation only reverts when both conditions are true (`block.timestamp > endTime && !goalReached`), which means the modifier will NOT revert even when the end time has also finished and the goal has also been reached.

Location of Affected Code

File: contracts/DaosLive.sol

Recommendation

Modify the condition to revert when either the end time has passed OR the goal has been reached, ensuring the funding phase is properly enforced in all cases.

```
 modifier onlyFundingPhase() {
-    if (block.timestamp > endTime && !goalReached) {
+     if (block.timestamp > endTime || goalReached) {
         revert OnlyFundingPhase();
      }
     _;
  }
```

Team Response

Fixed.

[L-05] Vesting Schedule Can Be Manipulated by DAO Manager

Severity

Low Risk

Description

The DAO Manager can manipulate vesting schedules by setting arbitrary start and end dates, potentially locking users' tokens for longer than necessary. For example, setting the first vesting period (June-July) with 50% unlock, then second period (October-November) with the remaining 50%, effectively locking tokens for 3 extra months, by leaving 3 months of Gap between the 2nd start date of the 2nd index.

Location of Affected Code

File: contracts/DaosVesting.sol

Recommendation

Implement two validations in the vesting schedule: the first vesting period must start at the current block timestamp, and each subsequent period's start date must match the previous period's end date. This ensures continuous vesting without unnecessary lock periods.

Team Response

Fixed.

[L-06] Unrestricted Vesting End Date in Schedule Setting

Severity

Low Risk

Description

The DAO Manager can set extremely long vesting periods (e.g., 100 years) in the vesting schedule without any restrictions. This allows malicious DAO Managers to effectively lock users' tokens indefinitely, preventing them from ever claiming their tokens. As there is no check in the code, and the `daoManager` can be malicious, as this is not the protocol's trusted role.

Location of Affected Code

File: contracts/DaosVesting.sol

Recommendation

Implement a signature verification mechanism similar to the `contribute()` function, where the protocol verifier must sign off on the vesting schedule parameters. This ensures that vesting periods are reasonable and prevents malicious DAO Managers from setting excessive vesting lock periods.

Team Response

Fixed.

[L-07] Unfair Token Distribution Due to Discount Mechanism

Severity

Low Risk

Description

When some users contribute with discounts, they get a higher `effectiveValue` for their contribution, which increases the `totalEffectiveRaised`. This dilutes the token allocation for users who contributed without discounts, as the token distribution is based on `effectiveContributions` rather than actual contributions. For example, if User A contributes 1 ETH with a 0% discount and User B contributes 1 ETH with a 50% discount, User B will receive more tokens than User A despite contributing the same amount.

Location of Affected Code

File: contracts/DaosLive.sol

Recommendation

Implementing a separate token distribution mechanism that doesn't affect users who contribute without discounts.

Team Response

Fixed.

[I-01] Redundant NFT Receiver Functions in `DaosLive` Contract

Severity

Informational Risk

Description

The functions `onERC721Received()` and `onERC1155Received()` are implemented in the `DaosLive` contract, but they serve no practical purpose in the current context. The contract is not designed to receive NFTs via direct transfers, making these functions redundant and unnecessary. Their presence may introduce confusion or suggest unused functionality.

Location of Affected Code

File: contracts/DaosLive.sol

```
function onERC721Received(address, address, uint256, bytes calldata)
    external pure returns (bytes4) {
     return IERC721Receiver.onERC721Received.selector;
}

function onERC1155Received(address, address, uint256, uint256, bytes
    calldata) external pure returns (bytes4) {
     return IERC1155Receiver.onERC1155Received.selector;
}
```

Recommendation

Remove the `onERC721Received()` and `onERC1155Received()` functions from the `DaosLive` contract to reduce code bloat and improve clarity. If NFT support is not planned or required, these functions should not be included to maintain a minimal and purpose-driven codebase.

Team Response

Fixed.

[I-02] Missing Proper Time Validation in `create()` Function in `DaosFactory`

Severity

Informational Risk

Description

The `DaosFactory::create()` function only checks if `endTime` is in the past (`endTime < block.timestamp`), but doesn't validate that `endTime` should not be the present time. This could allow setting `endTime` to the current block timestamp, which might not be the intended behavior for fundraising campaigns.

Location of Affected Code

File: contracts/DaosFactory.sol

Recommendation

Add validation to ensure `endTime` is strictly in the future:

```
if (endTime <= block.timestamp) revert InvalidEndTime();
```

Team Response

Fixed.

[I-03] Redundant `receive()` Function in `DaosLive`

Severity

Informational Risk

Description

The `DaosLive` contract implements a `receive()` function that simply reverts without any custom error message. This is functionally equivalent to not having a `receive()` function at all, as in both cases it would not let any direct ETH transfers to the contract.

Location of Affected Code

File: contracts/DaosLive.sol

Recommendation

Remove the redundant `receive()` function, or at least add a custom error message that shows clarity for the revert.

Team Response

Fixed.

[I-04] Missing Strict Checks for Vesting Start Time Validation

Severity

Informational Risk

Description

In `getClaimablePercent()`, the function returns 0 only if `block.timestamp < schedule.date`, but not when they are equal. This means users can claim a minute amount of tokens exactly at the start time of a vesting period, even though no time has actually elapsed since the vesting began.

Location of Affected Code

File: contracts/DaosVesting.sol#L114

```
function getClaimablePercent( address dao, uint256 index ) public view
    returns (uint256) {
  Schedule memory schedule = schedules[dao][index];

  if (block.timestamp < schedule.date) return 0;
// code
}
```

Recommendation

Modify the condition to include the equal case (`block.timestamp <= schedule.date`) to ensure users can strictly claim tokens only after the vesting period has actually started and some time has elapsed.

Team Response

Fixed.

shieldify

Thank you!