C8NTINUUM Launchpad Audit Report

Version 1.0

IhtishamSudo

December 29, 2025

C8NTINUUM Launchpad

IhtishamSudo

December 29, 2025

Prepared by: IhtishamSudo

Auditor: - IhtishamSudo

Table of Contents

Protocol Summary

C8NTINUUM Launchpad is a decentralized token launch platform that enables users to create and participate in token pledges. The protocol facilitates token swaps between pledged tokens and graduated ERC20 tokens through integration with Uniswap V2. Additionally, it includes a Rewards system for staking, vesting, and task-based token distribution.

Disclaimer

IhtishamSudo makes all effort to find as many vulnerabilities in the code in the given time period. This security audit by IhtishamSudo is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the implementation of the contracts.

Risk Classification

Likelihood	Impact	Severity
High	Critical	C
High	High	H
High	Medium	H/M
High	Low	M
Medium	Critical	H
Medium	High	H/M
Medium	Medium	M
Medium	Low	M/L
Low	Critical	M
Low	High	M
Low	Medium	M/L
Low	Low	L

Audit Details

Scope

The following contracts were in scope for the audit:

- `contracts/Launchpad.sol`
- `contracts/Pledge.sol`
- `contracts/rewards/Rewards.sol`
- `contracts/rewards/Stake.sol`
- `contracts/rewards/Vesting.sol`
- `contracts/rewards/Tasks.sol`
- `contracts/uniswapV2/UniswapV2Router02.sol`
- `contracts/uniswapV2/libraries/UniswapV2Library.sol`

Roles

- **Owner**: Has administrative privileges including setting fees, allocating balances, and withdrawing funds
- **Creator**: The deployer of a pledge who receives creator supply and volume fees
- **User**: Can pledge tokens, unpledge, swap tokens, and claim rewards

Executive Summary

The audit identified 9 issues across various severity levels. The most critical findings relate to potential denial-of-service attacks on the Pledge contract and incorrect token routing that could result in permanent loss of user funds.

Issues found

Severity	Count
Critical	2
High	2
Medium	3

Severity	Count
Low	1
Informational	1
Total	**9**

Findings

Critical

[Crit-1] Attacker Can Permanently Brick Pledge Contract by Force-Sending as Low as 1 Wei via `selfdestruct`

Description

The `pledge()` function in `Pledge.sol` calculates the valid contribution amount by subtracting the contract's excess balance from the user's `msg.value` inside an `unchecked` block:

```
function pledge(uint amountOutMin, address to) external payable
    onlyLaunchpad returns(uint amountOut) {
     if (pair != address(0x0)) revert Launched();

     uint amountIn = msg.value;
     uint currentBalance = address(this).balance;

     if (currentBalance > graduationCtm) {
         unchecked {
             uint remainder = currentBalance - graduationCtm;
             amountIn -= remainder;  // @audit: underflow if remainder >
                 msg.value

             uint percentage = amountIn * 1e18 / msg.value;
             amountOutMin = amountOutMin * percentage / 1e18;
         }
         // Pay back the rest
         (bool sent, ) = payable(to).call{value: msg.value - amountIn}("
            ");
         if (!sent) revert CouldNotSend(0);
     }
     // ...
}
```

An attacker can exploit this by force-sending ETH to the Pledge contract via `selfdestruct`. Since `selfdestruct` bypasses the contract's receive/fallback functions and cannot be blocked, the attacker can push the contract's balance above the `graduationCtm` threshold.

When a legitimate user then calls `pledge()` with `msg.value`, the calculation becomes: - `currentBalance` = forced balance + `msg.value` (e.g., `graduationCtm + 1 wei + msg.value`) - `remainder` = `currentBalance - graduationCtm` = `msg.value + 1 wei` - `amountIn -= remainder` → underflows because `remainder > msg.value`

Inside the `unchecked` block, this underflow wraps around to a massive value (~2^256 - 1), causing subsequent revert, effectively bricking the contract.

Impact

- **Permanent DoS**: The pledge contract becomes permanently unusable for all users.

Proof of Concepts

1. Pledge contract is deployed with `graduationCtm = 8888 ether`
2. Current contract balance is 8000 ether (CTM) (close to graduation)
3. Attacker deploys malicious contract with 889 ether and calls `selfdestruct(pledgeAddress)`
4. Pledge contract balance is now 8889 ether (CTM) (> graduationCtm)
5. User calls `pledge()` via Launchpad with `msg.value = 1 ether` (CTM)
6. Inside `pledge()`:
    - `currentBalance` = 8890 ether (CTM)
    - `remainder` = 8890 - 8888 = 2 ether (CTM)
    - `amountIn` = 1 ether (CTM) - 2 ether (CTM) → underflows in `unchecked` block
7. function's next execution revert - causing pledge to fail

Recommended Mitigation

Use an internal accounting variable to track the CTM balance instead of relying on `address(this).balance` directly:

```
+ uint public ctmBalance;

function pledge(uint amountOutMin, address to) external payable
    onlyLaunchpad returns(uint amountOut) {
     if (pair != address(0x0)) revert Launched();

     uint amountIn = msg.value;
-    uint currentBalance = address(this).balance;
+    ctmBalance += msg.value;
```

```
+   uint currentBalance = ctmBalance;

    if (currentBalance > graduationCtm) {
        unchecked {
            uint remainder = currentBalance - graduationCtm;
            amountIn -= remainder;
+           ctmBalance -= remainder;

            uint percentage = amountIn * 1e18 / msg.value;
            amountOutMin = amountOutMin * percentage / 1e18;
        }
        // Pay back the rest
        (bool sent, ) = payable(to).call{value: msg.value - amountIn}("
            ");
        if (!sent) revert CouldNotSend(0);
    }
    // ...
}
```

This way, force-sent ETH via `selfdestruct` does not affect the internal accounting.

Team Response

Fixed

[Crit-2] `swapExactTokensForTokens` Third Branch Sends Output Tokens to Contract Instead of User, Permanently Locking Funds

Description

The `swapExactTokensForTokens` function in `Launchpad.sol` handles three different swap scenarios based on which tokens have active pledges. The first two branches correctly route tokens to the user-specified `to` address:

- **Branch 1** (both tokens have active pledges): Unpledges the first token to receive ETH at the contract, then pledges that ETH into the second token which is sent directly to `to`.
- **Branch 2** (only second token has active pledge): Swaps a graduated ERC20 for ETH via the router to `address(this)`, then pledges that ETH into the second token, sending tokens to `to` via the pledge mechanism.

However, the **third branch** (only first token has active pledge):

```
} else if(isPledgeActiveFirst) {
    // The case where we unpledge to buy an ERC20
```

```
3       amountOut = IPledge(pledges[firstToken]).unpledge(_msgSender(),
            amountIn, 1,address(this));
4
5       address[] memory sliced = new address[](2);
6       for (uint i = 0; i < 2; i++) {
7           sliced[i] = path[i + 1];
8       }
9
10      uint[] memory amountsOut = router.swapExactETHForTokens{value:
            amountOut}(amountOutMin, sliced, address(this), deadline); //
            @audit: address(this) should be `to`
11      amountOut = amountsOut[1];
12  }
```

The function correctly unpledges the first token to receive ETH at the contract, then swaps that ETH for a graduated ERC20 via `router.swapExactETHForTokens`. The problem is that the recipient parameter is set to `address(this)` instead of `to`, meaning the output tokens are sent to the Launchpad contract instead of the user.

Impact

- **Permanent Loss of User Funds**: The output tokens from the swap are sent to the Launchpad contract instead of the intended recipient (`to`). The Launchpad contract has no mechanism to withdraw arbitrary ERC20 tokens, meaning these funds are permanently locked.

Proof of Concepts

1. User holds TokenA (has active pledge) and wants to swap to TokenB (graduated, no active pledge)
2. User calls `swapExactTokensForTokens(100, minOut, [TokenA, WETH, TokenB], userAddress, deadline)`
3. The function enters the third branch (`isPledgeActiveFirst == true`, `isPledgeActiveSecond == false`)
4. User's 100 TokenA is unpledged for ETH at the contract
5. ETH is swapped for TokenB via the router, but sent to `address(this)` (Launchpad)
6. User's `to` address receives nothing
7. TokenB is now stuck in the Launchpad contract with no way to retrieve it

Recommended Mitigation

Change the recipient parameter from `address(this)` to `to` in the `swapExactETHForTokens` call:

```
- uint[] memory amountsOut = router.swapExactETHForTokens{value:
    amountOut}(amountOutMin, sliced, address(this), deadline);
+ uint[] memory amountsOut = router.swapExactETHForTokens{value:
    amountOut}(amountOutMin, sliced, to, deadline);
```

Team Response

Fixed

High

[High-1] `withdrawToken` Allows Owner to Withdraw Allocated Tokens

Description

The `withdrawToken` function in `Rewards.sol` allows the owner to transfer any ERC20 tokens out of the contract without updating any internal accounting variables:

```
function withdrawToken(address _token, address _to, uint _amount)
    external onlyOwner {
    IERC20(_token).safeTransfer(_to, _amount);
}
```

The contract maintains separate state variables to track allocated tokens: - `stakeTokenBalance` - tokens reserved for staking rewards - `tasksTokenBalance` - tokens reserved for task rewards - `lockedVestingTokenBalance` - tokens locked in vesting schedules

These sum to `allocatedTokenBalance()`:

```
function allocatedTokenBalance() public view returns(uint) {
    return stakeTokenBalance + tasksTokenBalance +
        lockedVestingTokenBalance;
}
```

The `unallocatedTokenBalance()` function relies on the invariant that actual token balance equals allocated plus unallocated:

```
function unallocatedTokenBalance() public view returns(uint) {
    return token.balanceOf(address(this)) - allocatedTokenBalance();
}
```

When `withdrawToken` executes, it transfers tokens via `safeTransfer` but never decrements the accounting state. There is no validation that withdrawn amounts are limited to truly unallocated tokens.

Impact

- **Protocol Insolvency**: If the owner withdraws tokens that are allocated to users (stakers, vesters, task reward recipients), the contract becomes insolvent while its internal state claims everything is properly funded.

Proof of Concepts

1. Contract holds 1,000,000 tokens with the following allocation:
    - `stakeTokenBalance` = 500,000
    - `tasksTokenBalance` = 300,000
    - `lockedVestingTokenBalance` = 100,000
    - `allocatedTokenBalance()` = 900,000
    - `unallocatedTokenBalance()` = 100,000
2. Owner calls `withdrawToken(token, ownerAddress, 1000000)` - withdrawing all tokens
3. Actual token balance is now 0, but:
    - `stakeTokenBalance` still equals 500,000
    - `tasksTokenBalance` still equals 300,000
    - `lockedVestingTokenBalance` still equals 100,000
4. `unallocatedTokenBalance()` now reverts due to underflow (0 - 900,000)
5. Stakers, vesters, and task claimants cannot withdraw their entitled tokens
6. Protocol is insolvent - users are owed 900,000 tokens that don't exist

Recommended Mitigation

Add a balance check to `withdrawToken` that prevents withdrawal of allocated tokens:

```
function withdrawToken(address _token, address _to, uint _amount)
    external onlyOwner {
+    if (_token == address(token)) {
+        require(_amount <= unallocatedTokenBalance(), "Not enough
    unlocked token balance");
+    }
     IERC20(_token).safeTransfer(_to, _amount);
}
```

Team Response

Fixed

[High-2] `pledgeCreationFee` Permanently Locked in Launchpad Contract Due to Missing Withdrawal Mechanism

Description

The `Launchpad.sol` contract collects ETH (CTM) through multiple channels but provides no mechanism to retrieve the accumulated funds.

The contract accepts ETH (CTM) in two ways:

1. **Pledge Creation Fee**: The `launchPledge()` function is marked `payable` and enforces a mandatory fee:

```
function launchPledge(LaunchParams memory params) external payable
    returns (address token, address pledge) {
     require(msg.value == pledgeCreationFee, "Must pay pledge creation
        fee");
     // ... rest of function
}
```

Where `pledgeCreationFee` defaults to 10 ETH (CTM). Every successful pledge creation deposits this amount directly into the Launchpad contract.

2. **Direct Transfers**: The empty `receive()` fallback function allows anyone to send ETH (CTM) directly to the contract:

```
receive() external payable {}
```

Other functions do not accumulate ETH in the contract: - `swapExactETHForTokens()` forwards `msg.value` directly to the Pledge contract - `launch()` passes `msg.value` straight to the Uniswap router via `addLiquidityETH{value: msg.value}()`

However, there is **no withdrawal function** for native ETH (CTM) in the Launchpad contract. The contract has no `withdraw()`, `rescueETH()`, or similar function with `transfer()`, `send()`, or `call{value:}()` to extract accumulated fees.

Impact

- **Permanent Fund Lock**: Every pledge creation permanently locks 10 CTM in the contract with no way to recover it.

Proof of Concepts

1. User A calls `launchPledge()` with `msg.value = 10 CTM` → Launchpad balance = 10 CTM
2. User B calls `launchPledge()` with `msg.value = 10 CTM` → Launchpad balance = 20 CTM
3. All CTM is permanently locked in the contract

Recommended Mitigation

Option 1: Implement a controlled withdrawal function to enable fee recovery:

```
+ function withdrawETH(address payable _to, uint _amount) external
    onlyOwner {
+     require(_amount <= address(this).balance, "Insufficient balance")
    ;
+     (bool success, ) = _to.call{value: _amount}("");
+     require(success, "ETH transfer failed");
+ }
```

Option 2: Modify `launchPledge()` to immediately forward fees to a designated treasury address rather than holding them in the contract:

```
+ address public treasury;

function launchPledge(LaunchParams memory params) external payable
    returns (address token, address pledge) {
    require(msg.value == pledgeCreationFee, "Must pay pledge creation
        fee");
+   (bool success, ) = treasury.call{value: msg.value}("");
+   require(success, "Fee transfer failed");
    // ... rest of function
}
```

Team Response

Fixed

Medium

[Med-1] `allocateTasksBalance` Uses Wrong Modifier for CTM Allocation, Allowing Over-Allocation of CTM Funds

Description

The `allocateTasksBalance` function in `Rewards.sol` allows the owner to reserve a specified amount of tokens or CTM (native currency) for task rewards by increasing `tasksTokenBalance` or `tasksCtmBalance`. The function uses a boolean `_isCtm` parameter to determine whether to allocate ERC20 tokens or native CTM.

However, the function only applies the `enoughTokenBalance` modifier regardless of which asset type is being allocated:

```
function allocateTasksBalance(bool _isCtm, uint _amount) external
    payable onlyOwner enoughTokenBalance {
     // Optimistically add and check if we overflow
     if (_isCtm) {
         tasksCtmBalance += _amount;
     } else {
         tasksTokenBalance += _amount;
     }
}
```

The `enoughTokenBalance` modifier validates against the ERC20 token balance:

```
modifier enoughTokenBalance() {
     _;
     if (allocatedTokenBalance() > token.balanceOf(address(this)))
        revert NotEnoughTokenBalance();
}
```

When `_isCtm == true`, the function increases `tasksCtmBalance` but validates against ERC20 token balance instead of native CTM balance. The correct modifier `enoughCtmBalance` exists but is not applied:

```
modifier enoughCtmBalance() {
     _;
     if (allocatedCtmBalance() > address(this).balance) revert
        NotEnoughCtmBalance();
}
```

Impact

- **Over-allocation of CTM**: The owner can allocate more CTM to the tasks balance than the contract actually holds, as the validation check is performed against the wrong asset type.

Proof of Concepts

1. Contract has 100 ERC20 tokens and 10 CTM (native currency)
2. Owner calls `allocateTasksBalance(true, 1000)` to allocate 1000 CTM for tasks
3. The `enoughTokenBalance` modifier checks: `allocatedTokenBalance()> token.balanceOf(address(this))`
4. If `allocatedTokenBalance()` is still less than the 100 ERC20 tokens held, the check passes
5. `tasksCtmBalance` is now set to 1000, but the contract only holds 10 CTM
6. When users try to claim CTM task rewards, the transfers will fail due to insufficient native balance

Recommended Mitigation

Apply the correct modifier based on the asset type being allocated. The function should use `enoughCtmBalance` when allocating CTM:

```
- function allocateTasksBalance(bool _isCtm, uint _amount) external
    payable onlyOwner enoughTokenBalance {
+ function allocateTasksBalance(bool _isCtm, uint _amount) external
    payable onlyOwner enoughTokenBalance enoughCtmBalance {
     // Optimistically add and check if we overflow
     if (_isCtm) {
         tasksCtmBalance += _amount;
     } else {
         tasksTokenBalance += _amount;
     }
}
```

By adding both modifiers, the function will validate against both balance types.

Team Response

Fixed

[Med-2] Anyone Can Corrupt Contributors List by Invoking `unpledge` with Zero Amount, Griefing Legitimate Users

Description

The `swapExactTokensForETH` function in `Launchpad.sol` forwards `amountIn` directly to `Pledge.unpledge` without validating that it's greater than zero:

```
function swapExactTokensForETH(uint amountIn, uint amountOutMin,
    address[] calldata path, address to, uint deadline) external
    nonReentrant returns(uint amountOut) {
     if (block.timestamp > deadline) revert Expired();
     if (path[1] != WETH) revert IncorrectPath(1);
     // @audit: No check for amountIn > 0
     amountOut = IPledge(pledges[path[0]]).unpledge(_msgSender(),
         amountIn, amountOutMin, to);
}
```

The `unpledge` function in `Pledge.sol` also doesn't validate for zero amounts and has a flaw in the contributor removal logic:

```
function unpledge(address from, uint amountIn, uint amountOutMin,
    address to) external onlyLaunchpad returns(uint amountOut) {
     if (pair != address(0x0)) revert Launched();

     uint balance = balanceOf[from];
```

```
5       if (balance < amountIn) revert NotEnoughBalance();  // 0 < 0 is
            false, passes!
6
7       // ... calculations happen with amountIn = 0 ...
8
9       balanceOf[from] = balance;  // Still 0
10
11      if (balance == 0) {
12          uint index = indexOf[from];  // Returns 0 for non-contributors
                (default value)
13          address last = contributors[contributors.length - 1];
14
15          contributors[index] = last;  // Overwrites contributors[0]!
16          indexOf[last] = index;
17
18          contributors.pop();
19          delete indexOf[from];
20      }
21      // ...
22  }
```

When `amountIn == 0` and `from` is not a real contributor: 1. `balance = balanceOf[from]` = 0 (default value) 2. `balance < amountIn` → `0 < 0` is false, so the check passes 3. No actual balance changes occur 4. `balance == 0` triggers the contributor removal branch 5. `indexOf[from]` returns 0 (default for non-existent mapping entry) 6. The legitimate contributor at index 0 gets overwritten/removed!

Impact

- **Contributors List Corruption**: Attacker can remove legitimate contributors from the list without affecting their actual balances.

Proof of Concepts

1. Pledge contract has contributors: `[Alice, Bob, Charlie]` at indices `[0, 1, 2]`
2. Attacker calls `swapExactTokensForETH(0, 0, [token, WETH], attacker, deadline)`
3. `unpledge` is called with `from = attacker`, `amountIn = 0`
4. `balanceOf[attacker]` = 0, check `0 < 0` passes
5. `balance == 0` branch executes:
    - `index = indexOf[attacker]` = 0 (default value)
    - `last = contributors[2]` = Charlie
    - `contributors[0] = Charlie` (Alice overwritten!)
    - `indexOf[Charlie] = 0`
    - `contributors.pop()` removes Charlie from index 2

6. Contributors list is now: `[Charlie, Bob]` - Alice is gone!

Recommended Mitigation

Apply multiple fixes:

1. In `Launchpad.sol`, require `amountIn > 0`:

```
function swapExactTokensForETH(uint amountIn, uint amountOutMin,
    address[] calldata path, address to, uint deadline) external
    nonReentrant returns(uint amountOut) {
     if (block.timestamp > deadline) revert Expired();
     if (path[1] != WETH) revert IncorrectPath(1);
+    require(amountIn > 0, "Amount must be greater than 0");
     amountOut = IPledge(pledges[path[0]]).unpledge(_msgSender(),
         amountIn, amountOutMin, to);
}
```

2. In `Pledge.sol`, ensure `amountIn > 0` in `unpledge`:

```
function unpledge(address from, uint amountIn, uint amountOutMin,
    address to) external onlyLaunchpad returns(uint amountOut) {
     if (pair != address(0x0)) revert Launched();
+    require(amountIn > 0, "Amount must be greater than 0");

     uint balance = balanceOf[from];
     if (balance < amountIn) revert NotEnoughBalance();
     // ...
}
```

3. In `Pledge.sol`, verify user was actually a contributor before removal:

```
     balanceOf[from] = balance;

-    if (balance == 0) {
+    if (balance == 0 && indexOf[from] != 0 || (indexOf[from] == 0 &&
    contributors.length > 0 && contributors[0] == from)) {
         uint index = indexOf[from];
         address last = contributors[contributors.length - 1];
         // ...
     }
```

Or more cleanly, track contributor status with a separate mapping:

```
+ mapping(address => bool) public isContributor;

// In pledge():
if (balanceOf[to] == 0) {
+    isContributor[to] = true;
     indexOf[to] = contributors.length;
     contributors.push(to);
```

```
}

// In unpledge():
- if (balance == 0) {
+ if (balance == 0 && isContributor[from]) {
+   isContributor[from] = false;
    // ... removal logic
}
```

Team Response

Fixed

[Med-3] Fee-On-Transfer Tokens Break Swap Logic in Launchpad, Causing Transaction Reverts

Description

The `swapExactTokensForTokens` function in `Launchpad.sol` assumes the amount of tokens received is exactly equal to the `amountIn` parameter when handling the second branch (only second token has active pledge):

```
} else if (isPledgeActiveSecond) {
    // The case where we use an ERC20 to pledge
    address[] memory sliced = new address[](2);
    for (uint i = 0; i < 2; i++) {
        sliced[i] = path[i];
    }

    IERC20(firstToken).safeTransferFrom(_msgSender(), address(this),
        amountIn);
    IERC20(firstToken).approve(address(router), amountIn);
    // Amount out min does not matter as the user sets it for the
        pledge and it will check it there
    uint[] memory amountsOut = router.swapExactTokensForETH(amountIn,
        1, sliced, address(this), deadline);
    // ...
}
```

For Fee-On-Transfer (FOT) tokens, a percentage of the transferred amount is deducted as a fee during the transfer. This means: - User transfers `amountIn` tokens - Contract receives `amountIn - fee` tokens (less than `amountIn`) - Contract approves and tries to swap `amountIn` tokens - Swap fails because contract only holds `amountIn - fee` tokens

Impact

- **Transaction Reverts**: Any swap involving FOT tokens as the first token in this branch will fail.

Proof of Concepts

1. User wants to swap 1000 FOT tokens (2% fee) for a pledged token
2. User calls `swapExactTokensForTokens(1000, minOut, [FOT, WETH, pledgedToken], user, deadline)`
3. Function enters the `isPledgeActiveSecond` branch
4. `safeTransferFrom` executes: user sends 1000, contract receives 980 (20 taken as fee)
5. `approve(router, 1000)` - approves 1000 tokens
6. `swapExactTokensForETH(1000, ...)` - tries to swap 1000 tokens
7. Router attempts to transfer 1000 tokens from Launchpad
8. Transaction reverts - Launchpad only has 980 tokens!

Recommended Mitigation

Update the logic to calculate the actual amount of tokens received by measuring the balance before and after the transfer:

```
} else if (isPledgeActiveSecond) {
    address[] memory sliced = new address[](2);
    for (uint i = 0; i < 2; i++) {
        sliced[i] = path[i];
    }

+   uint balanceBefore = IERC20(firstToken).balanceOf(address(this));
    IERC20(firstToken).safeTransferFrom(_msgSender(), address(this),
        amountIn);
-   IERC20(firstToken).approve(address(router), amountIn);
-   uint[] memory amountsOut = router.swapExactTokensForETH(amountIn,
    1, sliced, address(this), deadline);
+   uint amountReceived = IERC20(firstToken).balanceOf(address(this)) -
    balanceBefore;
+
+   IERC20(firstToken).approve(address(router), amountReceived);
+   uint[] memory amountsOut = router.swapExactTokensForETH(
    amountReceived, 1, sliced, address(this), deadline);

    amountOut = IPledge(pledges[secondToken]).pledge{value: amountsOut
        [1]}(amountOutMin, to);
}
```

Team Response

Fixed Partially

Low

[Low-1] Use `Ownable2Step` Instead of `Ownable` to Prevent Accidental Loss of Ownership

Description

The contracts `Launchpad.sol` and `Rewards.sol` inherit from OpenZeppelin's `Ownable` contract:

```
// Launchpad.sol
contract Launchpad is Ownable, ReentrancyGuard {
    constructor(...) Ownable(_msgSender()) { ... }
}

// Rewards.sol
contract Rewards is Ownable, Stake, Vesting, Tasks {
    constructor(address _token, address _creator) Ownable(_creator) {
        ... }
}
```

The standard `Ownable` contract allows the owner to transfer ownership in a single step via `transferOwnership(newOwner)`. If the owner accidentally transfers ownership to an incorrect address (typo, wrong address, zero address), the contract becomes permanently ownerless with no way to recover.

OpenZeppelin provides `Ownable2Step` which implements a two-step ownership transfer process: 1. Current owner calls `transferOwnership(pendingOwner)` to propose a new owner 2. Pending owner must call `acceptOwnership()` to confirm and complete the transfer

This prevents accidental transfers to addresses that cannot accept ownership.

Impact

- **No Recovery Mechanism**: Once ownership is transferred to a wrong address, there's no way to regain control.

Proof of Concepts

1. Owner intends to transfer ownership to `0x1234...5678`
2. Owner accidentally calls `transferOwnership(0x1234...5679)` (off-by-one typo)
3. Ownership immediately transfers to the wrong address
4. The correct address `0x1234...5678` cannot claim ownership
5. Contract is permanently ownerless - all admin functions locked

Recommended Mitigation

Replace `Ownable` with `Ownable2Step` in both contracts:

```
- import "@openzeppelin/contracts/access/Ownable.sol";
+ import "@openzeppelin/contracts/access/Ownable2Step.sol";

- contract Launchpad is Ownable, ReentrancyGuard {
+ contract Launchpad is Ownable2Step, ReentrancyGuard {
    constructor(...) Ownable(_msgSender()) { ... }
}

- contract Rewards is Ownable, Stake, Vesting, Tasks {
+ contract Rewards is Ownable2Step, Stake, Vesting, Tasks {
    constructor(address _token, address _creator) Ownable(_creator) {
        ... }
}
```

Team Response

Fixed

Informational

[Info-1] `multipleTaskClaims` Missing Array Length Validation

Description

The `multipleTaskClaims` function in `Rewards.sol` accepts five parallel arrays but does not validate that all arrays have equal lengths:

```
function multipleTaskClaims(
    address[] calldata _beneficiaries,
    bool[] calldata _isCtm,
    uint[] calldata _amounts,
    uint[] calldata _taskIds,
    bytes[] calldata _signatures
) public {
    for(uint i = 0; i < _beneficiaries.length; i++) {
        claimTask(_beneficiaries[i], _isCtm[i], _amounts[i], _taskIds[i
            ], _signatures[i]);
    }
}
```

The loop iterates based on `_beneficiaries.length` only, without checking if `_isCtm`, `_amounts`, `_taskIds`, and `_signatures` arrays have matching lengths.

Impact

- **Silent Data Truncation**: If any array is shorter than `_beneficiaries`, the transaction will revert with an out-of-bounds error.

Proof of Concepts

1. User/frontend accidentally calls:

```
multipleTaskClaims(
    [addr1, addr2, addr3],     // 3 elements
    [true, false],              // 2 elements - mismatch!
    [100, 200, 300],            // 3 elements
    [1, 2, 3],                  // 3 elements
    [sig1, sig2, sig3]          // 3 elements
)
```

2. Loop runs for `i = 0, 1, 2` based on `_beneficiaries.length`
3. At `i = 2`, accessing `_isCtm[2]` reverts with index out-of-bounds

Recommended Mitigation

Add array length validation at the start of the function:

```
function multipleTaskClaims(
    address[] calldata _beneficiaries,
    bool[] calldata _isCtm,
    uint[] calldata _amounts,
    uint[] calldata _taskIds,
    bytes[] calldata _signatures
) public {
+   require(
+       _beneficiaries.length == _isCtm.length &&
+       _beneficiaries.length == _amounts.length &&
+       _beneficiaries.length == _taskIds.length &&
+       _beneficiaries.length == _signatures.length,
+       "Array lengths must match"
+   );
    for(uint i = 0; i < _beneficiaries.length; i++) {
        claimTask(_beneficiaries[i], _isCtm[i], _amounts[i], _taskIds[i
            ], _signatures[i]);
    }
}
```

Team Response

Fixed