BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

DATE: 13th JAN 2025

PREPARED FOR: DARK MACHINE

About BlockApex

Founded in early 2021 BlockApex is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

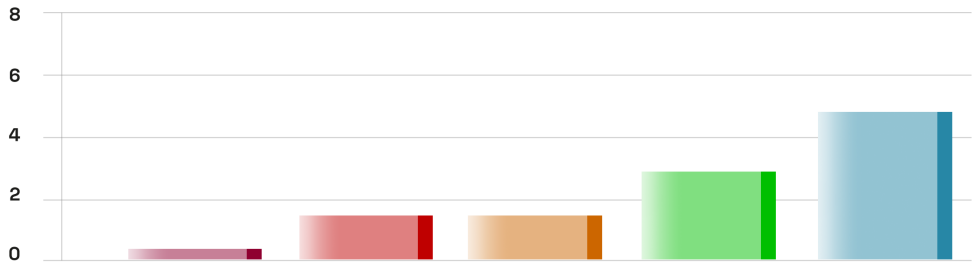
1 Executive Summary

Our team performed a Filtered Audit on the Dark Machine smart contracts, where two auditors independently conducted a detailed manual review of the code to identify potential vulnerabilities and logic flaws. This was followed by an automated tool-based analysis to detect common security issues. All flagged findings were manually verified and re-tested to eliminate false positives and confirm their validity.

Developer Response

Fixed	6
Acknowledged	4
Closed	0

Issues Overview



Critical
Critical vulnerabilities are too easy to exploit and can lead to damages/loss in assets or manipulations.

High
High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution.

Medium
Medium-level vulnerabilities are equally imperative to fix but they tend to have minimal impact on asset loss or data manipulations.

Low
Lowest-level vulnerabilities, informational errors, violating code styles/ practices usually can't affect smart contract execution; hence they can be ignored.

Info
Concerns that have no direct negative impact. These may help improve coding practices.

1.1 Scope

1.1.1 In Scope

Overview of the Contracts: Dark Machine is a decentralized staking and reward distribution protocol that integrates NFT staking, ERC20 token staking, vesting, and soft-staking mechanisms into a unified ecosystem. The platform is designed to incentivize user participation through dynamic reward phases, flexible staking options, and fair reward distribution. The smart contracts audited are deployed on EVM-compatible blockchains and focus on secure staking operations, reward management, and access control. The protocol allows users to stake tokens and NFTs to earn rewards, while administrators manage staking pools and reward phases.

Contracts in Scope:

1. DMG7Q1.sol – Standard NFT contract for minting and managing NFTs.
2. DMVIPA.sol – NFT contract with tracking mechanisms for ownership duration and staking.
3. MXNA.sol – ERC20 token contract with minting, burning, and transfer restrictions.
4. SaleTokenClaim.sol – Contract for managing token vesting, claiming schedules, and reward distribution.
5. SoftStaking.sol – Contract for soft-staking NFTs and distributing reward points based on ownership duration.
6. TokenStaking.sol – ERC20 staking contract with dynamic reward phases and reward distribution logic.

Initial Commit Hash: 4baed88edf38c58ba395397c4143108abb9e43ae

Final Commit Hash: 937f0d7cb8f595b34214630eefc23728ad1ce1ba

1.1.2 Out of Scope

Any features, contracts, or functionalities not explicitly mentioned in the “In Scope” section are considered out of scope for this audit. This includes any external integrations, off-chain components, or third-party smart contracts interacting with the Dark Machine protocol.

1.2 Methodology

The codebase was audited using a Filtered Audit technique. A band of two (2) auditors scanned the codebase in an iterative process for a time spanning 2 weeks. Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation. Furthermore, the audit progressed with manual code reviews to identify logical flaws, potential security vulnerabilities, and areas for code optimization. This phase was complemented by evaluations of software and security design patterns, code styling, and adherence to best practices. Automated tools were also utilized to flag common vulnerabilities, and all findings were manually verified to eliminate false positives and ensure accuracy.

Project Overview

Dark Machine is a decentralized staking and reward distribution protocol designed to seamlessly integrate NFT staking, ERC20 token staking, dynamic reward distribution, and soft-staking mechanisms into a unified ecosystem. It emphasizes flexibility, transparency, and user incentivization while leveraging modular smart contract architecture to support diverse staking models. Key Component Include:

1. **Token Staking Module:**
    - Dynamic Reward Phases: Supports phased reward distribution with customizable start and end times, enabling precise control over token emission schedules.
    - Flexible Lock Periods: Users can stake tokens for variable durations, with longer lock periods offering increased reward multipliers.
2. **NFT Staking Module:**
    - Soft Staking Mechanism: NFT holders can earn points based on ownership duration and the number of NFTs held.
    - Dynamic Point Multipliers: Users are incentivized to hold more NFTs for higher multipliers and increased rewards.
    - Customizable Point Distribution Windows: Admins can define specific time frames for point accumulation.
3. **Vesting & Claiming Module:**
    - Token Claiming: Users can claim vested tokens based on predefined schedules, including cliff periods and weekly unlock amounts.
    - TGE (Token Generation Event): Initial tokens are released immediately at TGE, with the remaining tokens following a vesting schedule.
    - Unlockable Amounts: Claim calculations ensure fairness, considering elapsed time and vesting conditions.
4. **Governance & Access Control:**
    - Role-Based Access Control: Administrative roles such as DEFAULT_ADMIN_ROLE and OPERATOR_ROLE regulate sensitive operations.

This architecture ensures efficient staking, dynamic reward allocation, and robust administrative controls, offering a scalable and transparent staking ecosystem for token and NFT holders alike.

Centralization & Administrative Roles

Most protocols require a degree of centralization to function effectively, often involving roles like “super admins” with elevated privileges for managing critical operations. In the case of this protocol, roles such as DEFAULT_ADMIN_ROLE and OPERATOR_ROLE are specifically defined to handle administrative actions. While we trust the operators of the protocol to act rationally and responsibly, we want to highlight the actions these roles are capable of performing for transparency:

1. Enables the withdrawal of specific ERC20 tokens to a designated wallet address through withdrawTokens function.
2. Allows the withdrawal of all ERC20 tokens from the contract to a specified wallet through withdrawAllTokens function.
3. Facilitates the transfer of native tokens held by the contract to a chosen address through withdrawNativetokens function.

1.3 Questions for Security Assessment

These are the main questions for security assessment, but the evaluation is not limited to these alone.

1. How does the protocol ensure efficient and bounded reward calculations in staking pools to prevent gas exhaustion during large-scale staking and reward distribution?

2. What safeguards are implemented to prevent unauthorized access to critical administrative functions, such as modifying pool parameters or withdrawing tokens?

3. How does the contract handle staking and unstaking operations to ensure that tokens are not transferred before proper validation of user balances and lock durations?

4. Are there mechanisms in place to prevent reentrancy attacks, especially in functions involving token transfers and reward claims?

5. Does the contract verify the availability of reward tokens during pool creation to ensure sufficient funds for promised reward distribution?

6. How does the contract handle edge cases in reward distribution, such as preventing users with shorter lock durations from unfairly accumulating long-term rewards?

7. Are there proper validations to ensure that administrative roles cannot misuse withdrawal functions to drain staking or reward funds?

8. Does the contract prevent the possibility of overflow or underflow in reward calculations, particularly during reward accumulation over time?

9. How does the protocol ensure that staking tokens and reward tokens cannot be mistakenly or maliciously withdrawn through general withdrawal functions?

10. Are fallback mechanisms in place to handle failed transactions during reward distribution or staking operations to maintain the protocol's integrity?

1.4 Status Descriptions

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

1.5 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	HIGH	Flawed Access Control on 'Blocked' and 'Not Allowed Addresses' for MXNA Token	FIXED
#2	MEDIUM	Unrestricted Configuration in setPool Can Lead to Unintended Outcomes	PARTIALLY FIXED
#3	LOW	Unfair Reward Distribution Due to Staking Lock Discrepancy	ACKNOWLEDGED
#4	LOW	Self-Transfer of Single NFT Resets Ownership Timestamp, Disrupting Rewards	ACKNOWLEDGED
#5	LOW	Lack of Reward Fund Validation During Pool Creation	ACKNOWLEDGED
#6	INFO	Unsafe ERC20 Operations in TokenStaking and SaleTokenClaim Contracts	FIXED
#7	INFO	Missing Event Emissions for State Changes	FIXED
#8	INFO	Unsafe Use of _mint() Instead of _safeMint() for ERC721 Tokens	ACKNOWLEDGED
#9	INFO	Costly Operations in Loops and Unbounded Growth of userStakeIds	FIXED
#10	INFO	Missing Validation and Logical Checks	FIXED

2 Findings and Risk Analysis

2.1 Flawed Access Control on 'Blocked' and 'Not Allowed Addresses' for MXNA Token

Severity: High

Status: Fixed

Location :

contracts/MXNA.sol

Description The MXNA token contract fails to enforce restrictions on blocked and not allowed users effectively, leading to the following issues:

1. Minting for Blocked Users: The mint function allows blocked addresses to receive MXNA tokens, which undermines the concept of blocking.
2. Burning by Blocked Users: Blocked users can burn their MXNA tokens, which should not be permitted.
3. Unrestricted Transfers: Blocked and unauthorized users can freely transfer MXNA tokens, bypassing restrictions, even when transfers are disabled.

Proof of Concept

```
function test_blocked_user_transfer() public {
    // Disable transfers globally
    token.setTransferEnabled(false);

    // **Not Allowed User Test Cases**

    // Mint tokens to a not allowed address. This operation is allowed as it&#x27;s an admin action.
    token.mint(not_Allowed_address, 1000 ether); // Minting 1000 MXNA tokens to a not allowed user
    assertEq(token.balanceOf(not_Allowed_address), 1000 ether); // Balance of not allowed user should be updated

    // Not allowed users can burn tokens, which should ideally be restricted
    vm.prank(not_Allowed_address); // Simulate action as the not allowed address
    token.burn(10 ether); // Burn 10 MXNA tokens
    assertEq(token.balanceOf(not_Allowed_address), 990 ether); // Balance should be reduced after burning

    // Not allowed users can transfer tokens between themselves even when transfers are disabled
    vm.prank(not_Allowed_address); // Simulate action as the not allowed address
    token.transfer(address(1), 90 ether); // Transfer 90 MXNA tokens to another address
    assertEq(token.balanceOf(not_Allowed_address), 900 ether); // Balance should reflect the transfer

    // **Blocked User Test Cases**

    // Add the address to the blocked list
    token.addBlockedAddress(blocked_user);

```

```
26      // Blocked users can still receive tokens via minting, which should be restricted
27      token.mint(blocked_user, 1000 ether); // Minting 1000 MXNA tokens to a blocked user
28      assertEq(token.balanceOf(blocked_user), 1000 ether); // Balance of blocked user should
            be updated
29
30      // Blocked users can burn tokens, which should not be allowed
31      vm.prank(blocked_user); // Simulate action as the blocked user
32      token.burn(10 ether); // Burn 10 MXNA tokens
33      assertEq(token.balanceOf(blocked_user), 990 ether); // Balance should be reduced after
            burning
34
35      // Allow a specific user for transfers
36      token.addAllowedAddress(allowed_user);
37
38      // Blocked users can transfer tokens to allowed users, and allowed users can transfer
            tokens to blocked users,
39      // even when transfers are disabled. This behavior is unintended.
40      vm.prank(blocked_user); // Simulate action as the blocked user
41      token.transfer(allowed_user, 90 ether); // Transfer 90 MXNA tokens to an allowed user
42      assertEq(token.balanceOf(blocked_user), 900 ether); // Balance should reflect the
            transfer
43
44      // Enable transfers globally
45      token.setTransferEnabled(true);
46
47      // Blocked users can transfer tokens to anyone when transfers are enabled, bypassing
            their blocked status
48      vm.prank(blocked_user); // Simulate action as the blocked user
49      token.transfer(address(1), 100 ether); // Transfer 100 MXNA tokens to another address
50      assertEq(token.balanceOf(blocked_user), 800 ether); // Balance should reflect the
            transfer
51
52      // Admin assigns a blocked user to the allowed list, which effectively removes
            restrictions from the blocked user
53      token.addAllowedAddress(blocked_user); // Blocked user is now also an allowed user
54  }
```

Recommendation Update the _beforeTokenTransfer function to strictly enforce transfer rules for blocked and unauthorized users.

2.2 Unrestricted Configuration in setPool Can Lead to Unintended Outcomes

Severity: Medium

Status: Partially Fixed

Location :

/contracts/TokenStaking.sol

Description The setPool function allows admins to update critical pool parameters such as staking and reward tokens, start and end times, and reward phases even after the pool is live. This level of control introduces risks, having following impact:

1. **Unexpected Token Changes:** Altering the staking or reward tokens mid-pool disrupts existing stakes and rewards logic, leading to mismatched tokens for users during staking or withdrawal
2. **Timing Manipulations:** Adjusting `startTime` or `endTime` can result in incorrect reward calculations, skipped rewards, or premature pool closure, negatively affecting stakers.
3. **Phase Modifications:** Changing reward phases (e.g., adding, removing, or modifying them) post-initialization can break reward distribution consistency and user expectations. Reward Reductions: Reducing totalRewards or modifying phaseRewards after the pool starts may cheat stakers of promised rewards.

Recommendation :

1. Prevent changes to stakingToken and rewardToken once the pool has started to ensure token consistency.
2. Disallow modifications to startTime after the pool begins; ensure any endTime changes can only extend the pool duration.
3. Validate that lastRewardTime is properly updated whenever startTime or endTime is modified to maintain reward accuracy.
4. Lock totalRewards and phaseRewards once the pool has started to avoid altering user expectations.
5. Restrict modifications to reward phases.
6. Validate reward phases to confirm they are sequential, non-overlapping, and aligned with the pool's overall duration.
7. Enforce a rule preventing reductions to endTime after the pool begins, allowing only extensions to support ongoing stakes.

```
    IERC20 _rewardToken,
    uint256 _startTime,
    uint256 _endTime,
    uint256[] memory _phaseStartTimes,
    uint256[] memory _phaseEndTimes,
    uint256[] memory _phaseRewards
```

```
) external onlyRole(OPERATOR_ROLE) {
    Pool storage pool = pools[_poolId];
    // Ensure pool has not started
    require(block.timestamp &lt; pool.startTime, &#x27;Cannot modify active pool&#x27;);
    // Prevent token changes after initialization
    require(pool.startTime == 0, &#x27;Token updates not allowed after start&#x27;);
    // Validate end time modifications
    require(_endTime &gt;= pool.endTime, &#x27;End time cannot be reduced&#x27;);
    require(_endTime &gt; _startTime, &#x27;End time must be after start time&#x27;);
    // Validate reward phases
    require(
        _phaseStartTimes.length == _phaseEndTimes.length &amp;&amp; _phaseEndTimes.length
            == _phaseRewards.length,
        &#x27;Phase arrays must be same length&#x27;
    );
    uint256 totalRewards = 0;
    for (uint256 i = 0; i &lt; _phaseRewards.length; i++) {
        totalRewards += _phaseRewards[i];
    }
    // Lock existing reward phases after pool start
    require(poolRewardPhases[_poolId].length == 0 || block.timestamp &lt; pool.startTime,
        &#x27;Reward phases are immutable&#x27;);
    // Update pool data
    pool.totalRewards = totalRewards;
    pool.endTime = _endTime;
    delete poolRewardPhases[_poolId];
    for (uint256 i = 0; i &lt; _phaseStartTimes.length; i++) {
        require(_phaseEndTimes[i] &gt; _phaseStartTimes[i], &#x27;Phase end time must be
            after start time&#x27;);
        require(
            _phaseStartTimes[i] &gt;= _startTime &amp;&amp; _phaseEndTimes[i] &lt;=
                _endTime,
            &#x27;Phase times must be within pool times&#x27;
        );
        if (i &gt; 0) {
            require(_phaseStartTimes[i] == _phaseEndTimes[i - 1], &#x27;Phases must be
                consecutive&#x27;);
        }
        RewardPhase memory phase = RewardPhase({
            startTime: _phaseStartTimes[i],
            endTime: _phaseEndTimes[i],
            totalRewards: _phaseRewards[i]
        });
        poolRewardPhases[_poolId].push(phase);
    }
    emit PoolSet(
        _poolId,
        pool.stakingToken,
        pool.rewardToken,
        totalRewards,
        pool.startTime,
        _endTime,
        _phaseStartTimes,
        _phaseEndTimes,
        _phaseRewards
    );
}
```

2.3 Unfair Reward Distribution Due to Staking Lock Discrepancy

Severity: Low

Status: Acknowledged

Location :

/contracts/TokenStaking.sol

Description In the current reward system, rewards are distributed evenly based on the amount staked, without properly considering the lock duration. For example, in a 4-week pool with two reward phases of 2 weeks each, if User 1 stakes for just 1 week and User 2 stakes for the full 4 weeks, both can still earn the same rewards if User 1 decides not to unstake after their lock ends. This creates an imbalance where User 2, who is locked and cannot withdraw funds for the full duration, receives no additional benefit compared to User 1, who had the flexibility to leave but chose to stay. This undermines the purpose of staking locks, making long-term commitments meaningless in terms of reward incentives.

Proof of Concept

```
function testParallelRewardsCheck() public {
        uint256 startTime = block.timestamp;
        uint256 endTime = startTime + 4 weeks;

        uint256[] memory phaseStartTimes = new uint256[](2);
        uint256[] memory phaseEndTimes = new uint256[](2);
        uint256[] memory phaseRewards = new uint256[](2);

        phaseStartTimes[0] = startTime;
        phaseEndTimes[0] = startTime + 2 weeks;
        phaseRewards[0] = 2000 ether;

        phaseStartTimes[1] = startTime + 2 weeks;
        phaseEndTimes[1] = endTime;
        phaseRewards[1] = 4000 ether;

        //creating pool 1
        vm.prank(operator);
        staking.addPool(
            IERC20(address(token)),
            IERC20(address(rewardToken)),
            startTime,
            endTime,
            phaseStartTimes,
            phaseEndTimes,
            phaseRewards
        );

        (, , , uint256 totalRewards, , , , , , ) = staking.pools(1);
        assertEq(totalRewards, 6000 ether, &#x27;Total rewards mismatch&#x27;);

        //staking 1000 tokens for a week
        vm.startPrank(user1);
        token.approve(address(staking), 10000 ether);
        staking.stake(1, 1000 ether, 1);
        assertEq(token.balanceOf(user1), 0);
        vm.stopPrank();
```

```
38
39          //staking 1000 tokens for 4 weeks
40          vm.startPrank(user2);
41          token.approve(address(staking), 10000 ether);
42          staking.stake(1, 1000 ether, 4);
43          assertEq(token.balanceOf(user2), 0);
44          vm.stopPrank();
45
46          //after 4 weeks
47          vm.warp(block.timestamp + 4 weeks);
48
49          vm.startPrank(user1);
50          staking.claim(1);
51          staking.unstake(1);
52          console.log((rewardToken.balanceOf(user1)) / 1e18);
53          console.log((token.balanceOf(user1)) / 1e18);
54          vm.stopPrank();
55
56          vm.startPrank(user2);
57          staking.claim(2);
58          staking.unstake(2);
59          console.log((rewardToken.balanceOf(user1)) / 1e18);
60          console.log((token.balanceOf(user1)) / 1e18);
61          vm.stopPrank();
62      }
```

Result

```
1  Logs:
2    User1 Reward:        2926
3    User1 staked amount: 1000
4    User2 Reward:        3073
5    User2 staked amount: 1000
```

Recommendation The reward system should factor in lock durations when distributing rewards. Rewards should either scale based on lock commitment or finalize once a lock period ends to prevent unintended accumulation. If the lock period is not necessary for the rewards calculation then it should be imposed on any user.

2.4 Self-Transfer of Single NFT Resets Ownership Timestamp, Disrupting Rewards

Severity: Low

Status: Acknowledged

Location :

contract/DMVIPA.sol

Description In the DMVIPA NFT contract, the _beforeTokenTransfer hook updates the firstOwnedTime mapping to track the first time an address owns an NFT. However, the current implementation incorrectly resets the firstOwnedTime for an address during self-transfers (i.e., from == to). This happens due to the condition:

```
if (from != address(0) &amp;amp;&amp;amp; balanceOf(from) == 1) {
    firstOwnedTime[from] = 0;
}
```

The condition assumes that the sender (from) no longer holds any NFTs, but in self-transfers, the ownership remains unchanged.

Impact Self-transfers reset firstOwnedTime, causing reward calculations in the SoftStaking contract to exclude the affected NFT.

Proof of Concept

```
function test_firstOwnedTime_field_reset() public {
    nft.mint(user1);
    console.log(nft.getFirstOwnedTime(user1)); // Logs correct timestamp
    vm.prank(user1);
    nft.transferFrom(user1, user1, 0);         // Self-transfer
    console.log(nft.getFirstOwnedTime(user1)); // Logs 0 (incorrect)
}
```

Recommendation Update the _beforeTokenTransfer logic to exclude self-transfers from resetting firstOwnedTime.

```
if (from != address(0) &amp;amp;&amp;amp; balanceOf(from) == 1 &amp;amp;&amp;amp; from != to) {
    firstOwnedTime[from] = 0;
}
```

2.5 Lack of Reward Fund Validation During Pool Creation

Severity: Low

Status: Acknowledged

Location :

1. /contracts/TokenStaking.sol
2. /contract/SaleTokenClaim.sol

Description When an admin creates a pool using the addPool function, there is no validation to ensure that the reward tokens (rewardToken) corresponding to the declared totalRewards are actually transferred to the contract. Additionally, the admin should transfer the totalAmount of MXNA tokens to the SaleTokenClaim contract, ensuring they are locked. At the time of creating the ClaimInfo for users, the corresponding amount of MXNA tokens should be available and locked.

Impact Stakers may not receive their promised rewards if the pool is underfunded.

Recommendation Ensure that the reward tokens are present in the contract at the time of pool creation by adding a validation check in the addPool function.

```
require(
    _rewardToken.balanceOf(address(this)) &gt;= totalRewards,
    &quot;Insufficient reward tokens in contract&quot;
);
```

2.6 Unsafe ERC20 Operations in TokenStaking and SaleTokenClaim Contracts

Severity: Info

Status: Fixed

Location :

1. /contracts/TokenStaking.sol
2. /contract/SaleTokenClaim.sol

Description The TokenStaking and SaleTokenClaim contracts use low-level ERC20 functions (transfer, transferFrom) without verifying return values. This can lead to unexpected behavior with non-standard tokens. It is recommended to use OpenZeppelin's SafeERC20 library, which ensures robust error handling for token transfers and mitigates risks associated with unsafe ERC20 operations.

Impact Failure to use SafeERC20 operations can result in untracked token losses, low-level failures or inconsistencies.

Recommendation Replace all unsafe ERC20 operations with SafeERC20 library functions from OpenZeppelin.

- Use safeTransfer, safeTransferFrom, and safeApprove for token operations.
- For native ETH transfers, ensure the use of safe patterns like Address.sendValue or proper require checks to handle transfer failures.

2.7 Missing Event Emissions for State Changes

Severity: Info

Status: Fixed

Location :

1. /contracts/DMG7Q1.sol
2. /contract/DMVIPA.sol
3. /contract/SaleTokenClaim.sol
4. /contracts/TokenStaking.sol

Description Several functions in the contracts modify state variables without emitting events. Emitting events is essential for providing transparency, enabling off-chain monitoring, and facilitating debugging. This omission reduces visibility into critical contract operations and hinders efficient tracking of state changes.

Recommendation Emit appropriate events whenever state variables are updated to ensure traceability and transparency.

2.8 Unsafe Use of _mint() Instead of _safeMint() for ERC721 Tokens

Severity: Info

Status: Acknowledged

Location :

1. /contracts/DMG7Q1.sol
2. /contract/DMVIPA.sol

Description The contracts utilize the _mint() function from the ERC721 standard to mint tokens. However, _mint() does not ensure that the recipient address is capable of handling ERC721 tokens, potentially resulting in tokens being permanently locked in contracts or addresses that cannot handle ERC721 tokens. It is recommended to use _safeMint() instead, as it includes safety checks for recipient compatibility.

Recommendation Replace all occurrences of _mint() with _safeMint() to ensure that the recipient address implements the ERC721Receiver interface and is capable of handling ERC721 tokens.

2.9 Costly Operations in Loops and Unbounded Growth of userStakeIds

Severity: Info

Status: Fixed

Location :

1. /contracts/DMG7Q1.sol
2. /contract/DMVIPA.sol
3. /contract/SaleTokenClaim.sol
4. /contracts/TokenStaking.sol

Description Costly SSTORE operations occur within loops, increasing gas usage and risking out-of-gas errors. Additionally, the userStakeIds mapping grows linearly as users stake, leading to high gas costs for iteration or related operations.

Impact :

1. Excessive gas consumption and potential out-of-gas errors when operating on large data sets.
2. Increasing gas costs for operations involving userStakeIds as the array grows linearly with user staking activities

Recommendation :

1. Optimize loops by using local variables for intermediate results.
2. Remove stake IDs from userStakeIds during unstaking to prevent unchecked growth.

2.10 Missing Validation and Logical Checks

Severity: Info

Status: Fixed

Description Several areas across the contracts lack necessary validation checks, which could lead to unintended behavior or improper configurations. Key issues include missing timestamp comparisons, inadequate parameter validation, and logical constraints not being enforced. **Instances & Recommendation**

1. **setFirstOwnedTime() File:** DMVIPA.sol **Issue:** Missing check to ensure time >= block.timestamp. **Recommendation:** Validate time to ensure it is not in the past.
2. **setClaimStartTime() File:** SaleTokenClaim.sol **Issues:**
   - _claimStartTime should be greater than block.timestamp.
   - Should only allow this to be set once.

   Recommendation: Add checks to enforce these constraints.
3. **setClaimInfo() File:** SaleTokenClaim.sol **Issues:**
   - tgePercentage must not exceed 100.
   - vestingDuration and cliffDuration must not be 0.
   - cliffDuration must not exceed vestingDuration.

   Recommendation: Implement these validations.
4. **setPointStartTime() File:** SaleTokenClaim.sol **Issue:** Missing check to ensure _pointStartTime >= block.timestamp. **Recommendation:** Add validation for the timestamp comparison.
5. **addPool() File:** TokenStaking.sol **Issues:**
   - _phaseStartTimes, _phaseEndTimes, and _phaseRewards must have lengths greater than 0.
   - _startTime must be greater than block.timestamp.

   Recommendation: Enforce these checks.
6. **setPool() File:** TokenStaking.sol **Issues:**
   - _poolId must be less than nextPoolId.
   - Do not allow changes if the pool has already started (_startTime < block.timestamp).

- _phaseStartTimes, _phaseEndTimes, and _phaseRewards must have lengths greater than 0.

Recommendation: Add these validations.

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts