



SMART CONTRACT SECURITY

V 1.0

DATE: 10 JAN 2024

PREPARED FOR: ECLIPSE FINANCE

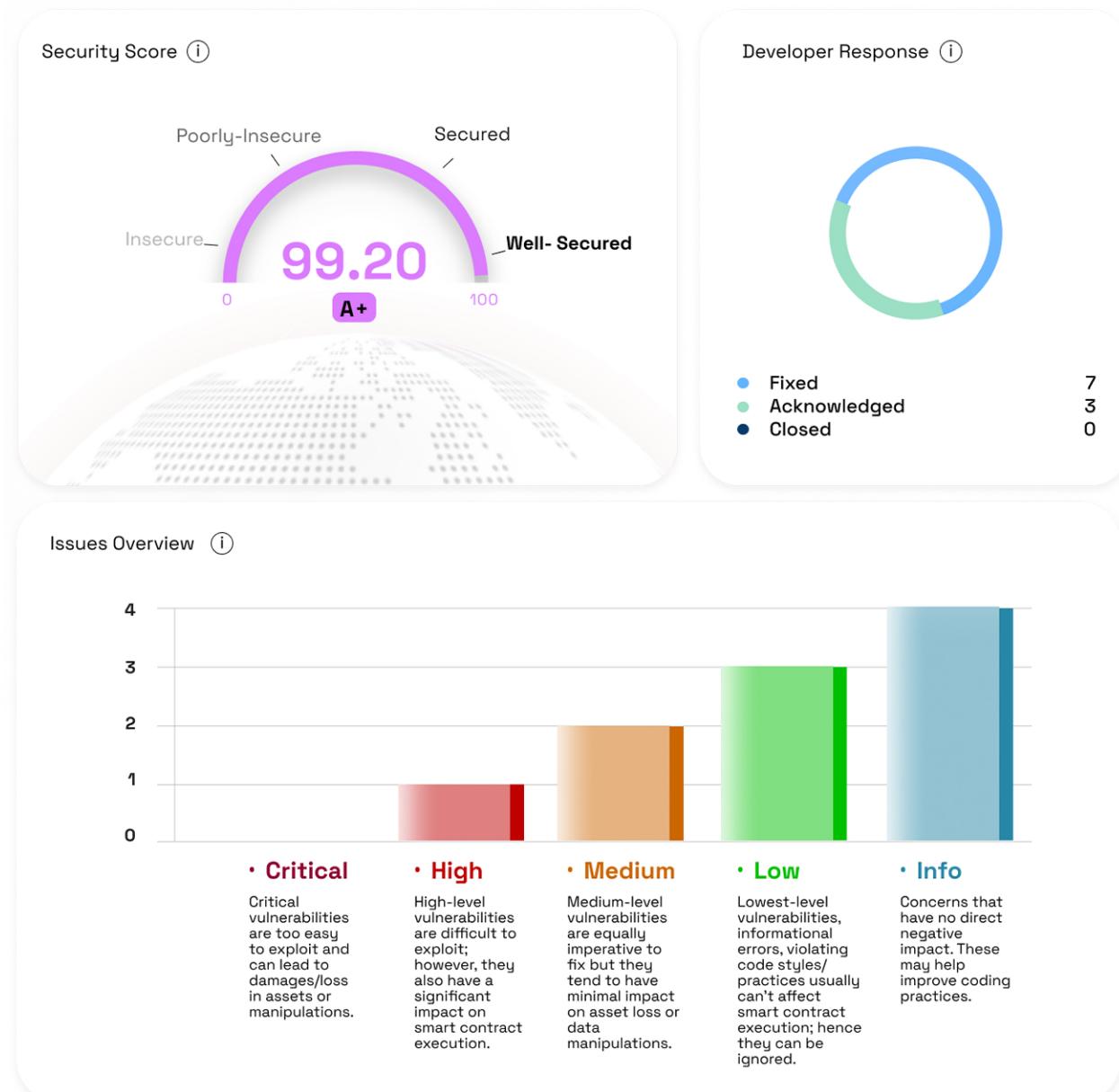


Contents

1 Executive Summary	3
1.1 Scope	4
1.1.1 In Scope	4
1.1.2 Out of Scope	4
1.2 Methodology	4
2 Summary of Findings	6
3 Findings and Risk Analysis	7
3.1 Forced Retention of User Funds in Multi-Tier Locking Dynamics	7
3.2 Unbounded Gas Consumption Risk Due to Excessive Stake/lock Entries	8
3.3 Risk of Temporary Locked Funds Due to Staking Token Modification	9
3.4 Lack of Validation for Excessive Lock Amounts	11
3.5 Missing validation of TokenFactory Denom in the Contract	12
3.6 Discrepancy Between Cosmic Essence Calculation and Documentation	13
3.7 Absence of 48-Hour Unstaking Period	14
3.8 Missing Validation for Zero Amount Withdrawal	15
3.9 Lack of NatSpec Documentation	16
3.10 Redundant Vault Storage in Tier Relocking Process	17

1 Executive Summary

Our team performed a technique called **Filtered Audit**. A thorough and rigorous manual testing process involving line by line code review for bugs was carried out. All the raised flags were manually reviewed and re-tested to identify any false positives. The figures presented herein offer a comprehensive bird's-eye perspective on the protocol's security.



1.1 Scope

1.1.1 In Scope

Overview of Staking Contract: The Staking Contract under audit is a comprehensive smart contract designed for the Cosmos platform, primarily focusing on staking functionalities. It serves as a decentralized finance (DeFi) application allowing users to stake tokens, manage locked funds, and earn rewards. Key functionalities include a robust reward distribution system based on staking duration and amount, and a flexible locking mechanism with various schedules for enhanced rewards. The contract also addresses early withdrawal penalties, ensuring commitment to lock periods, and incorporates essence accumulation, reflecting staking duration and volume.

Folder in scope: Contracts/staking_v2/

Initial Audit Commit Hash: [023c2e3104da7b2b6962f037e4d8a77d857570e2](#)

Fixed Commit Hash: [f5ec8aa77163e85e7eb0904e4ada9caa81b2d847](#)

1.1.2 Out of Scope

Any features or functionalities not explicitly mentioned in the “In Scope” section are also considered outside the scope of this audit.

1.2 Methodology

The codebase was audited using a filtered audit technique. A band of three (3) auditors scanned the codebase in an iterative process for a time spanning 4 days. 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/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles and best practices

Status Description

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.

2 Summary of Findings

S.No	Severity	Findings	Status
#1	HIGH	Forced Retention of User Funds in Multi-Tier Locking Dynamics	FIXED
#2	MEDIUM	Unbounded Gas Consumption Risk Due to Excessive Stake/lock Entries	ACKNOWLEDGED
#3	MEDIUM	Risk of Temporary Locked Funds Due to Staking Token Modification	FIXED
#4	LOW	Lack of Validation for Excessive Lock Amounts	FIXED
#5	LOW	Missing validation of TokenFactory Denom in the Contract	FIXED
#6	LOW	Discrepancy Between Cosmic Essence Calculation and Documentation	FIXED
#7	INFO	Absence of 48-Hour Unstaking Period	FIXED
#8	INFO	Missing Validation for Zero Amount Withdrawal	FIXED
#9	INFO	Lack of NatSpec Documentation	ACKNOWLEDGED
#10	INFO	Redundant Vault Storage in Tier Relocking Process	ACKNOWLEDGED

3 Findings and Risk Analysis

3.1 Forced Retention of User Funds in Multi-Tier Locking Dynamics

Severity: High

Status: Fixed

Location /src/execs.rs - `try_lock`, `try_unlock`

Description In the multi-tier locking structure, users encounter a challenge with locked-in funds when engaging with multiple tiers. Each tier in the contract has a distinct locking duration, and users are permitted to lock their funds across these tiers concurrently. However, an issue arises if a user wishes to unlock their funds after the completion of the locking period in one tier, while other tiers still have active locks. The contract's current mechanism does not allow for the release of funds from the completed tier independently. Instead, the unlock function applies penalties based on the tiers that are yet to complete their locking duration. This results in a situation where users' funds become effectively unusable in tiers with completed lock periods, as they are constrained by the longer-duration tiers.

Impact This challenge in the staking structure can lead to significant user inconvenience, especially for those who expect more control over their individual tier investments. The inability to independently withdraw from completed tiers without facing penalties from active ones can deter users from engaging in multi-tier staking. Furthermore, this could lead to confusion and a loss of trust, as users find their funds unexpectedly bound in the contract, undermining the flexibility and appeal of the tier-based staking approach.

Proof of Concept The issue manifests in scenarios where users have staked funds in multiple tiers with different lock periods. For example, a user who locks funds in tier 0 for 30 days and in tier 1 for 90 days cannot unlock from tier 0 without penalty once its period ends, due to the ongoing lock in tier 1.

Recommendation :

Contract Functionality Review: Re-examine the `try_unlock` function to consider allowing users to unlock funds from individual tiers upon completion of their respective locking periods, without penalties tied to other active tiers.

Enhance User Flexibility: Implement a mechanism that enables users to independently manage their stakes across different tiers, enhancing the staking system's user-friendliness and flexibility.

3.2 Unbounded Gas Consumption Risk Due to Excessive Stake/lock Entries

Severity: Medium

Status: Acknowledged

Location :

1. [src/execs.rs](#)
2. [src/math.rs](#)

Description This issue related to unbounded gas consumption, primarily due to the creation of multiple stake entries. When a user stakes or locks small units of tokens frequently, this results in an excessive number of stake entries written to the contract storage. The contract's calculation methods, which involve iterating over these numerous entries, can lead to significantly high gas consumption and ultimately lead to user specific Dos.

Impact :

In scenarios where excessive vaults are present, the implementation of unbounded loops in operations such as claiming rewards, locking, unlocking, and querying locked states could render the contract unusable due to potential out of gas errors.

Proof of Concept :

The issue arises from functions like `calc_essence_for_staking` and `calc_essence_for_locking_per_tier` that iterate over each vault entry to calculate staking essence and locking rewards. These calculations become increasingly gas-intensive as the number of stake entries grows. The risk is exacerbated when users frequently stake small amounts, creating a vast number of entries that the contract must process.

Recommendation :

Optimize Iteration Logic: Restructure the functions to minimize iterations over stake entries. This could involve aggregating similar stake entries or implementing more gas-efficient algorithms.

Limit Stake Entry Creation: Introduce mechanisms to limit the creation of excessive stake entries, such as minimum stake amounts or a cap on the number of stakes per user.:

Developer Response

The Dev intends to improve this in the future staking version.

3.3 Risk of Temporary Locked Funds Due to Staking Token Modification

Severity: Medium

Status: Fixed

Location :

1. src/instantiate.rs - try_instantiate
2. src/execs.rs - try_stake and try_unstake
3. src/execs.rs - try_update_config

Description Upon the instantiation of the staking contract, the default staking token is set to ECLIP_MAINNET, allowing users to stake these tokens. An admin function `try_update_config` permits changing the staking token. If this function is used to alter the staking token after users have already staked their funds, these users can no longer unstake their funds, as the `try_unstake` function checks for the current staking token in the contract, which is now different.

Impact This issue can lead to a temporary lock of user funds if the staking token is changed after staking. Users would be unable to unstake their funds until the admin resets the staking token to its original state.

Proof of Concept The following test case demonstrates the potential risk posed by changing the staking token in the staking contract, which could lead to user funds being locked. The scenario includes a user (Alice) staking tokens, an admin updating the staking token to a different one, and then Alice attempting to unstake her funds.

```
fn test_update_config_impact_on_unstaking() {
    // Constants for the test
    const AMOUNT: u128 = 10_000_000_000;
    let (mut app, contract_address) = init_app_and_users(AMOUNT);

    // Alice stakes her tokens
    app.execute_contract(
        Addr::unchecked(ALICE),
        contract_address.clone(),
        &ExecuteMsg::Stake {},
        &[coin(AMOUNT, ECLIP_TESTNET)],
    )
    .unwrap();

    // Query the initial configuration
    let initial_config: Config = app
        .wrap()
        .query_wasm_smart(contract_address.clone(), &QueryMsg::QueryConfig {})
        .unwrap();
    println!("Initial Config Details: {:?}", initial_config);

    // Admin updates the staking token in the contract's configuration
    app.execute_contract(
        Addr::unchecked(ADMIN),
        contract_address.clone(),
        &ExecuteMsg::UpdateConfig { new_staking_token: ECLIP_MAINNET },
        &[coin(AMOUNT, ECLIP_TESTNET)],
    )
    .unwrap();

    // Alice attempts to unstake her tokens
    app.execute_contract(
        Addr::unchecked(ALICE),
        contract_address.clone(),
        &ExecuteMsg::Unstake {},
        &[coin(AMOUNT, ECLIP_TESTNET)],
    )
    .unwrap_err();

    // Check that Alice's tokens are still in the contract
    let final_config: Config = app
        .wrap()
        .query_wasm_smart(contract_address.clone(), &QueryMsg::QueryConfig {})
        .unwrap();
    assert_eq!(final_config.staking_token, ECLIP_MAINNET);
}
```

```
&amp;ExecuteMsg::UpdateConfig {
    admin: None,
    staking_token: Some(&quot;token2&quot;).to_string(),
    lock_schedule: None,
    essence_per_second: None
},
&amp;[];
)
.unwrap();

// Query the configuration after the update
let updated_config: Config = app
    .wrap()
    .query_wasm_smart(contract_address.clone(), &amp;QueryMsg::QueryConfig {})
    .unwrap();
println!(&quot;Config AFTER Update: {:#?}&quot;, updated_config);

// Alice attempts to unstake her funds
app.execute_contract(
    Addr::unchecked(ALICE),
    contract_address.clone(),
    &amp;ExecuteMsg::Unstake {},
    &amp;[];
)
.unwrap();
}
```

In this test, after the staking token is updated by the admin, Alice's attempt to unstake her funds may fail due to the change in the staking token, potentially leading to her funds being locked within the contract. This scenario is particularly concerning if the new staking token does not hold sufficient liquidity or is not the token initially staked by users.

Recommendation :

Implement a mechanism that either prevents the staking token from being changed once staking operations have begun or remove this ability to update the staking has commenced

3.4 Lack of Validation for Excessive Lock Amounts

Severity: Low

Status: Fixed

Location The issue is present in the `try_lock` and `try_relock` function within the `execs.rs` file of the staking contract.

Description The lock function in the staking contract does not validate whether the amount a user wishes to lock or relock within the bounds of their currently bonded amount. This omission can lead to transaction failures without clear guidance for the user.

Recommendation Implement a validation check in the lock function to ensure the lock amount does not exceed the user's staked amount.

3.5 Missing validation of TokenFactory Denom in the Contract

Severity: Low

Status: Fixed

Location :

try_update_config , try_instantiate function in execs.rs

Description The `try_update_config` and `try_instantiate` function in the staking contract allows for the modification and initialization of the staking token address without thorough validation. This function accepts a new token address and updates the contract configuration without checking if the new address follows the expected format (`factory/neutron.../TOKEN_SUBDENOM`) or verifying its validity. Such oversight can result in incorrect token addresses being set, potentially disrupting the contract's operations.

Proof of Concept The issue lies in the implementation of the `try_update_config` and `try_instantiate` function, where the `staking_token` parameter can be updated to any string without validation:

```
if let Some(x) = staking_token {  
    config.staking_token = x;  
}
```

The lack of validation checks means that an admin could inadvertently or maliciously set an incorrect staking token address, leading to issues such as failed transactions, staking/unstaking operations using the wrong token, or even locking of user funds if the new token does not correspond to the actual staked tokens.

Recommendation :

1. **Regex Pattern Validation:** Integrate regex pattern matching for the `staking_token` parameter in the format "factory/NEUTON_ADDRESS/TOKEN_DENOM".
2. **NEUTON_ADDRESS Verification:** Use `deps.api.addr_validate(&x)?` to validate the NEUTON_ADDRESS within the token address.
3. **Format Standardization:** Ensure the token address strictly follows the 'factory/NEUTON_ADDRESS/TOKEN_DENOM' structure with correct placement of 'factory' and slashes.

3.6 Discrepancy Between Cosmic Essence Calculation and Documentation

Severity: Low

Status: Fixed

Location :

src/math.rs - calc_essence_for_staking

Description The staking contract's documentation claims that staking \$ECLIP tokens in a lockup vault generates Cosmic Essence at a 1:1 ratio with the staked amount over 12 months. However, a test case reveals a discrepancy. When 10,000,000,000 \$ECLIP tokens are staked for 30 days, it yields 82.19 essence instead of 83.3 , which does not align with the 1:1 ratio when projected over a year.

Proof of Concept The following test case demonstrates the issue:

```
#[test]
fn test_essence() {
    const AMOUNT: u128 = 10_000_000_000;
    const STAKING_PERIOD: u64 = 30 * 24 * 3600;
    let (mut app, contract_address) = init_app_and_users(AMOUNT);

    app.execute_contract(
        Addr::unchecked(ALICE),
        contract_address.clone(),
        &ExecuteMsg::Stake {},
        &[coin(AMOUNT, ECLIP_TESTNET)],
    ).unwrap();

    wait(&mut app, STAKING_PERIOD);

    let res: StakerInfoResponse = app
        .wrap()
        .query_wasm_smart(
            contract_address.clone(),
            &QueryMsg::QueryStakerInfo { staker: ALICE.to_string() },
        ).unwrap();
    println!("Essence: {}", res.essence);
}
```

Recommendation It is recommended to either update the documentation to accurately reflect the essence calculation as per the contract code or adjust the contract code to align with the documented 1:1 ratio over 12 months. This will ensure consistency and clarity for the users.

References Doc [Link](#)

3.7 Absence of 48-Hour Unstaking Period

Severity: Info

Status: Fixed

Location :

File: /src/execs.rs

Functions: [try_unstake](#)

Description As per the documentation of EclipseFi in Staking vaults "[The system requires a minimal 'time window'](#) of 48 hours to elapse before unstaking tokens, ensuring a brief commitment to the staked position. This is an invariant which is not implemented in the code, making it possible for users to unstake before the time window.

References <https://docs.eclipsefi.io/tokenomics-and-rewards/staking#staking-vaults>

3.8 Missing Validation for Zero Amount Withdrawal

Severity: Info

Status: Fixed

Location :

File: `execs.rs`

Function: `try_claim`

Description The staking contract lacks a validation check for zero-amount claims in the `try_claim` function. This function is designed to handle claim requests from users, including the calculation of the amount to be withdrawn based on claims and rewards. However, it currently does not validate whether the requested claimed amount is zero. Consequently, users may initiate transactions with a zero withdrawal amount.

Recommendation Implement an initial validation step in the `try_claim` function to check if the claimed amount is greater than zero. If the amount is zero, the function should return a custom error message, indicating that the claimed amount must be greater than zero to proceed.

3.9 Lack of NatSpec Documentation

Severity: Info

Status: Acknowledged

Location :

Examples include, but are not limited to, functions `try_lock`, `try_unlock`, `try_relock`, and `try_claim` in staking contract

Description The contract code is missing NatSpec comments, particularly in key functions like withdraw lock/unlock and bond/unbound. NatSpec (Natural Language Specification) provides essential documentation directly in the source code, aiding in understanding the function's purpose, input, output, and behavior

Impact Incorporate NatSpec comments throughout the contract, to clearly describe their functionalities. This practice will aid developers, auditors, and users in comprehending the intended behavior of the contract

Developer Response

This will be fixed in the future version of the staking contract

3.10 Redundant Vault Storage in Tier Relocking Process

Severity: Info

Status: Acknowledged

Location `src/execs.rs` - `try_lock`, `try_relock`

Description An inefficiency has been identified in the tier relocking process of a staking mechanism. When a user relocks from a lower tier (e.g., tier 0) to a higher tier (e.g., tier 4), the contract retains empty vaults from the previous tiers (0 to 3) instead of optimizing storage by removing or consolidating these now-redundant vaults. This behavior is evident from a test case (`test_total_vaults`) where, after relocking from tier 0 to tier 4, the user's information still displays empty vaults for tiers 0 to 3, alongside the active vault in tier 4.

Proof of Concept The `test_total_vaults` function demonstrates this issue. Initially, a user locks in tier 0. Upon relocking to tier 4, the query for locker vaults shows empty vaults for tiers 0 to 3 and the active vault only in tier 4. This is inefficient and could lead to unnecessary storage usage over time, especially as the number of users and transactions grows.

```
fn test_total_vaults () {
    const AMOUNT: u128 = 10_000_000_000;
    let (mut app, contract_address) = init_app_and_users(AMOUNT);

    // 1st Stake:
    // === 1st stake
    app.execute_contract(
        Addr::unchecked(ALICE),
        contract_address.clone(),
        &ExecuteMsg::Stake {},
        &[coin(AMOUNT, ECLIP_TESTNET)],
    )
    .unwrap();

    // 1st lock
    app.execute_contract(
        Addr::unchecked(ALICE),
        contract_address.clone(),
        &ExecuteMsg::Lock {
            amount: Uint128::new(10_000),
            lock_tier: 0,
        },
        &[], []
    )
    .unwrap();

    // Relocks from tier 0 to tier 4
    app.execute_contract(
        Addr::unchecked(ALICE),
        contract_address.clone(),
        &ExecuteMsg::Relock {
            amount: Uint128::new(10_000),
            from_tier: 0,
            to_tier: 4,
        },
    )
}
```

```
    &amp;[],  
)  
.unwrap();  
  
// query locker vaults:  
let res: StakerInfoResponse = app  
.wrap()  
.query_wasm_smart(  
    contract_address.clone(),  
    &amp;QueryMsg::QueryStakerInfo {  
        staker: ALICE.to_string(),  
    },  
)  
.unwrap();  
println!("Locker Vaults: {:?}", res.locker_infos,  
);  
}
```

Result:

```
Locker Vaults: [LockerInfo { lock_tier: 0, vaults: [] }, LockerInfo { lock_tier: 1, vaults: [] }, LockerInfo { lock_tier: 2, vaults: [] }, LockerInfo { lock_tier: 3, vaults: [] }, LockerInfo { lock_tier: 4, vaults: [Vault { amount: Uint128(10000), creation_date: 1000, claim_date: 1000 }] }]
```

Recommendation :

Optimize Vault Storage: Modify the `try_relock` function to automatically remove or consolidate vaults from lower tiers when a user relocks to a higher tier. This would ensure that only necessary and relevant vault information is stored and optimized gas

Developer Response

This will be fixed in the future version of the staking contract

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.