Quantstamp

Zero Staking

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	ERC20 and ERC721 Staking
Timeline	2024-05-20 through 2024-05-23
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Code Documentation
Source Code	• https://github.com/zer0-os/zModules #5f93318
Auditors	• Jeffrey Kam Auditing Engineer • Julio Aguilar Auditing Engineer • Jennifer Wu Auditing Engineer

Documentation quality	Medium
Test quality	High
Total Findings	10 Fixed: 6 Acknowledged: 4
High severity findings	2 Fixed: 2
Medium severity findings	2 Fixed: 2
Low severity findings	3 Fixed: 1 Acknowledged: 2
Undetermined severity findings	0
Informational findings	3 Fixed: 1 Acknowledged: 2

Summary of Findings

This audit focuses on a set of contracts that support lightweight staking for the Zero ecosystem where users can stake tokens to generate potential rewards. Additionally, there is a separate feature where the owner and operators can create and resolve game matches between players in the platform, facilitating the transfer of funds between the escrow contract and the users.

We found ten issues, two of which are high severity. The first issue allows an attacker to gain a disproportionate amount of rewards through a reentrancy vector in the stake and unstaking mechanism (ZS-1). The second issue is due to a lack of proper validation, which can cause a user to lose all his funds accidentally when unstaking through the exit mode (ZS-2).

Additionally, we found two medium-severity issues: users can receive fewer rewards due to integer division (ZS-3) and that staked funds can be lost due to a lack of separation between staked and reward funds (ZS-4). We recommend adding additional tests to validate fixes for high and medium issues identified.

We also included a few operational risks as low and informational severity issues. Users of the staking feature will greatly benefit from additional clarification from the team to help understand these risks when engaging with the project.

Regarding project quality, there is a lack of external-facing and technical documentation, so we have to rely on direct communication with the team to build our understanding. The test quality is decent, with branch coverage of around 79%. We believe the team can benefit from increasing the test coverage as well as having proper documentation to help users understand this new staking and matching features.

Update: The team has addressed all issues by either fixing or acknowledging them, as well as adding tests to validate the fixes. We appreciate the team's responsiveness and their thoroughness in their responses.

ID	DESCRIPTION	SEVERITY	STATUS
ZS-1	**Malicious User Can Drain Rewards Through Reentrancy in Staking**	• High	Fixed
ZS-2	**Loss of Pending Reward when Unstaking**	• High	Fixed
ZS-3	**Users Can Lose Out on Rewards if They Claim Too Frequently**	• Medium	Fixed

ID	DESCRIPTION	SEVERITY	STATUS
ZS-4	Mixing of Staked Tokens and Reward Tokens	Medium	Fixed
ZS-5	Compromised Owner or Operators Can Steal All Users' Escrowed Funds	Low	Acknowledged
ZS-6	There May Not Be Sufficient Funds in the Contract to Cover the Reward Claims	Low	Acknowledged
ZS-7	Missing Input Validation	Low	Fixed
ZS-8	User Can Stake a Small Amount Initially to Get an Early Unlock Timestamp	Informational	Acknowledged
ZS-9	Incompatible with Deflationary or Rebasing Tokens	Informational	Fixed
ZS-10	Privileged Roles and Ownership	Informational	Acknowledged

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

> **Disclaimer**
> Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

Repo: https://github.com/zer0-os/zModules(5f93318541de502e40d80354f1af21e6e8791f87)

Files: contracts/*

Files Excluded

Repo: https://github.com/zer0-os/zModules(5f93318541de502e40d80354f1af21e6e8791f87)

Files: contracts/mock/*

Findings

ZS-1
Malicious User Can Drain Rewards Through Reentrancy in Staking

High Fixed

Update

Marked as "Fixed" by the client.

Addressed in: `0a9a94bb49ec54fce5e6bd8859be3f981fbbac4a` .

File(s) affected: `StakingERC20.sol` , `StakingERC721.sol`

Description: The `StakingERC721.stake()` function allows a malicious staker to exploit the stale `staker.lastUpdatedTimestamp` due to a reentrancy vulnerability. During the staking process, the `_stake()` function calls `_safeMint()` , which calls `checkOnERC721Received()` . This allows the staker to re-enter the staking contract during the minting process. Since the `staker.lastUpdatedTimestamp` is only updated at the end of the `stake()` function, the staker can re-enter the `stake()` or `unstake()` function to trigger multiple `_checkRewards` calculations, thereby unfairly increasing their pending rewards based on the `staker.lastUpdatedTimestamp` .

Similar reentrancy risks exist in the `StakingERC20.stake()` function if ERC777 tokens are accepted as staking tokens. The ERC777 token standard was created to extend the capabilities available in ERC20 tokens and one of the features allows the ERC777 token contract to notify the sender and recipient when ERC777 tokens are sent or received, which a malicious staker can use to re-enter the staking function.

Recommendation: Implement reentrancy protection in the `stake()` and `unstake()` functions to prevent users from re-entering the functions and exploiting the stale `lastUpdatedTimestamp` . This can be achieved using a reentrancy guard, such as OpenZeppelin's `ReentrancyGuard` , or by updating the `staker.lastUpdatedTimestamp` earlier:

1. **Use Reentrancy Guard:** Add the `nonReentrant` modifier to the `stake()` and `unstake()` functions to prevent reentrancy.
2. **Update Timestamp Early:** Update the `staker.lastUpdatedTimestamp` before calling any external functions to ensure that the timestamp is accurate and up-to-date before any potential reentrant calls can occur.

ZS-2 Loss of Pending Reward when Unstaking

High Fixed

Update

Marked as "Fixed" by the client.

Addressed in: `2557a3905791ef0390513453c72077e2d7b0ed25` , `f4b922d5ddddb0b13753ad32bbcc5edb58aa27e7` .

The client provided the following explanation:

```
second commit for test fix
```

File(s) affected: `StakingERC20.sol` , `StakingERC721.sol`

Description: A user who unstakes his full staked balance with the flag `exit = true` can lose all his potential rewards because of the following lines:

```
// In StakingERC20.sol
if (staker.amountStaked - amount == 0) {
    delete stakers[msg.sender];
}
```

This effectively deletes the accounting of the rewards owed to the staker, causing the user to lose all his rewards even when `staker.owedRewards` is non-zero.

A similar vulnerability exists in `StakingERC721.sol` .

Recommendation: Ensure that pending rewards are properly handled even when the user's staker data is deleted. This can be achieved by either transferring the rewards immediately to the user, assuming `unlockTime` has passed, or maintaining a separate record of owed rewards that is not affected by the deletion of the Staker struct.

ZS-3
Users Can Lose Out on Rewards if They Claim Too Frequently

Medium | Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `391f5d7aef06ef430e4accf0edaca1f532275125`.
The client provided the following explanation:

> `The fix for the issue is in the StakingBase contract where the function ( _getPendingRewards() ) to calculate rewards was changed to prorate rewards between periods.`

Description: If a user calls `claim()` too frequently, he may lose out on rewards due to rounding calculation in `_getPendingRewards()`. In particular, when a user claims at times `A` and `B`, where the `B - A` is less than `periodLength`, then the rewards that should have been accumulated (i.e. `(rewardsPerPeriod * staker.amountStaked * ((B - A) / periodLength))` is lost simply due to rounding. Similarly, when a user claims and then stakes within a period's length, the partially accumulated rewards will be lost due to rounding.

Recommendation: Consider whether this is acceptable and intended. If not, consider updating the logic of `_getPendingRewards()` to ensure that any fractional rewards lost due to integer division are accounted for in future reward calculations. For example, one can update the function to the following:

```
function _getPendingRewards(
    Staker memory staker
) internal view returns (uint256) {
    uint256 flooredLastUpdateTimestamp = (staker.lastUpdatedTimestamp / periodLength) * periodLength;

    return
        staker.owedRewards +
        (rewardsPerPeriod *
            staker.amountStaked *
            ((block.timestamp - flooredLastUpdateTimestamp) /
                periodLength));
}
```

ZS-4 Mixing of Staked Tokens and Reward Tokens

Medium | Fixed

Update
Marked as "Fixed" by the client.
Addressed in: `78885e6851aa2c8f2a6892ef80937d7f53c1949c`. The client added a separate variable to track staked and unstaked funds accordingly through `totalStake`.

File(s) affected: `StakingERC20.sol`

Description: If the staked token and reward token are identical, the staked tokens may be incorrectly used to pay out rewards to other users. This occurs because there is no separation between the funds that are staked by users and the tokens that are sent to the contract to distribute as rewards. This lack of separation can lead to a situation where users' staked tokens are depleted to fulfill reward claims.

Furthermore, if the staked token and reward tokens are identical, the function `withdrawLeftoverRewards()` can be used to withdraw staked funds.

Recommendation: Implement internal accounting mechanisms to ensure a clear separation between staked tokens and reward tokens. This will prevent the unintended use of staked tokens for reward distribution.

ZS-5
Compromised Owner or Operators Can Steal All Users' Escrowed Funds

Low | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
Owner and operators of the smart contract will be under a secure ownership of Zero and WilderWorld.
Owner role will be assigned to a multisig wallet and Operator role will be assigned to the wallets
handled by the game itself. This access control was specifically made for the game software to use and
call contracts directly. Operators are there to not have a single point of failure by only allowing the
owner to call these functions, in addition to this, a game could possibly own and manage multiple
wallets for different use cases and game modes, these are reasons the logic allows for multiple
operators. We see it as a pretty common access control pattern for a smart contract based system.
```

File(s) affected: `Match.sol`,

Description: Suppose `A` is an address controlled by an owner (or operator). An owner (or operator) can create a match with high match fees and have one of the players being `A` and other players as victims. The owner (or operator) can then call `endMatch()` where the payout to `A` is the total of all the match fees from each player and every other player gets zero payouts, effectively stealing funds from all the other players.

Per communication with the team, the operators are addresses owned by the team, not some third parties, so the risk here is considered acceptable.

Recommendation: Consider whether the operators should have the privilege to do this. Furthermore, we would like to highlight the risks to potential users as this means funds in the Escrow is safe only if the owner and operators are not compromised.

ZS-6
There May Not Be Sufficient Funds in the Contract to Cover the Reward Claims

Low | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
This is a known issue and is acceptable. We have discussed this and are comfortable with it as long as
a user is able to see the amount of rewards available in a pool when they're staking or before they
claim. This information is made available by the StakingBase.sol contract with the
getContractRewardsBalance() function. In addition to this a contract will revert user's claim
transaction if the amount of funds stored on the contract is not enough to pay his rewards. For the
unstake() flow there is a flag exit that tells a contract that a user is ok with dropping their
rewards, in the case where they need to withdraw their stake while there are not enough rewards or in
any other case where a user would need to circumvent claiming or unstaking rules to exit whenever he
needs to.
```

File(s) affected: `StakingBase.sol`, `StakingERC20.sol`, `StakingERC721.sol`

Description: A user can only claim rewards from the contract if there are sufficient reward tokens in the staking contract. However, this heavily relies on the owner to provide funds to the contract to cover any user's claims. There may be times when there won't be sufficient funds in the contract to cover all user's claims.

Recommendation: Consider whether this is acceptable and document this clearly to the users.

ZS-7 Missing Input Validation

Low | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `4868d3571a60ce6f58b2e69377c62a01d59106af`, `ed64881c1c2a3a93b7b414736f5fcac1a6465f8c`.
The client has fixed all the items except for point 3.2, where they provided the following explanation:

```
3.2 “the list of players addresses should not contain duplicates; otherwise, a player can be charged
matchFee more than once."
  Looking for duplicates in an array inside of state changing function will be a pretty expensive
operation. Instead, we would prefer to rely on the game here and validate this on-chain. Since both
 startMatch()  and  endMatch()  require identical  players  arrays, a game would have to consider that
```

when it's building arguments to send to both these functions. We acknowledge this and decided to keep it as is.

It should also be noted that for 3.3, the client changed the game fee to be a fixed fee, therefore it is not possible to refund (e.g. end match) with zero game fee. The gameFee can be only changed by the contract owner which is a good separation of access control. The client confirmed that this is fine and is an intentional design.

Related Issue(s): SWC-123

Description: It is important to validate inputs, even if they only come from trusted addresses, to avoid human error. Specifically, in the following functions arguments of type `address` may be initialized with value `0x0`:

Following is the list of places that can potentially benefit from stricter input validation:

1. `StakingERC20`:
   1. the pending `rewards` of the `_baseClaim()` should not be zero.
   2. the `amount` of the `unstake()` should not be zero.
2. `StakingERC721`:
   1. the `tokenIds` length should be greater than zero and the `tokenIds` and `tokenUris` length should match in `stake()` function.
   2. the `tokenIds` length should be greater than zero in the `unstake()` function.
3. `Match`:
   1. the `matchFee` should be greater than zero in the `startMatch()` function; otherwise the `lockedFunds` will be zero and it will not be possible to end the match.
   2. the list of `players` addresses should not contain duplicates; otherwise, a player can be charged `matchFee` more than once.
   3. when ending a match via the `endMatch()` function, the authorized caller can specify any amount in `gameFee`. A compromised authorized party can set this value to be as much as 100%, stealing the users' payouts. Consider adding either a state variable with a fixed and known game fee or add an upper bound validation check.
4. `OwnableOperable`:
   1. when adding a new operator, the operator boolean should be previously `false` in the function `addOperator()`.
   2. when removing an existing operator, the operator boolean should be previously `true` in the function `removeOperator()`.

Recommendation: We recommend adding the relevant checks.

ZS-8
User Can Stake a Small Amount Initially to Get an Early Unlock Timestamp

Informational | Acknowledged

Update

Marked as "Acknowledged" by the client.
The client provided the following explanation:

```
This is a known issue but there isn't much to be done about it. If we update the user's unlockTimestamp
with every new stake, then we are forcing them to not add any additional stake to be able to claim
their rewards. They are able to game the system this way to get an earlier unlock time, but this isn't
meaningful in that they aren't able to claim additional rewards this way. The amount of rewards they
get are relative to what they've staked, so if they only stake a minimal amount, they will only be able
to claim minimal rewards even if they do game the system to get an earlier unlock timestamp. Any
significant stakes added later only begin accruing rewards at the time they are added, which means even
if they didn't game the system, it would take them the same amount of time to generate the same amount
of rewards.
  Another reason here is to not treat every staked amount as it's own separate entity in the contract's
state, significantly increasing gas usage and transaction cost as a result of all the logic and storage
usage this would require.
  The opposite is true in the unstaking case. The user can have a large portion accruing rewards and
then unstake all but 1 token in order to access the formerly staked funds without sacrificing their
rewards. A user can choose to do this, but their rewards going forward from that point in time will
only accrue at a minimal rate because they only have 1 token staked. And we do not consider it a
problem when the user who previously staked doesn't need to wait the timelock period again if he didn't
withdraw the full amount.
  We could enforce some minimum amount that has to be left in order to gain rewards, but no matter what
amount we choose users will game the system to always leave exactly as little as possible to do this
still.
```

File(s) affected: `StakingBase.sol`

Description: Since the unlock timestamp is never updated after the initial staking (see below in `_checkRewards()`), a user can game the system by staking an initial small amount (e.g. 1 token) to get an early `unlockTimestamp` and stake more as needed in the future.

```
function _checkRewards(Staker storage staker) internal {
    if (staker.amountStaked > 0) {
        // It isn't their first stake, snapshot pending rewards
        staker.owedRewards = _getPendingRewards(staker);
    } else {
        // Log the time at which this stake becomes claimable or unstakable
        // This is only done once per user
        staker.unlockTimestamp = block.timestamp + timeLockPeriod;
    }
}
```

Along the same line of reasoning, users could unstake all but 1 `wei` to bypass any unlock period in their next staking operation since the corresponding entry in the `stakers` mapping is not deleted.

Recommendation: Consider whether this is intended. If not, adjust the logic so that the unlock timestamp is updated accordingly upon receiving new stakes.

ZS-9 Incompatible with Deflationary or Rebasing Tokens

Informational | Fixed

Update

Marked as "Fixed" by the client.
Addressed in: `36a99cbbea137761d3265a79e275164237848544`. The team now added support for deflationary tokens.

File(s) affected: `StakingERC20.sol`, `Escrow.sol`

Description: The contracts are not designed to accommodate deflationary tokens when accepting token transfers. If a user transfers funds that incur a fee on transfer during order submission, the contract will store that the user has deposited more funds than they sent. This can cause issues during withdrawal or vault deposit.

Recommendation: Consider if this is acceptable. Otherwise, modify the logic to only account for the difference between the pre-transfer balance and the post-transfer balance of the contract. The difference will capture the amount of tokens that have been transferred regardless of the token transfer mechanism.

Update: Per communication with the team, we recommend only addressing the case of deflationary token by accounting the before and after balance. We currently do not recommend supporting rebasing token due to the complexities involved. If such a use case arises in the future, we recommend creating a new contract with the appropriate handling of rebasing tokens.

ZS-10 Privileged Roles and Ownership

Informational | Acknowledged

Update

Addressed in: `94df4a4edce471b95f287ec268cf0f1e1854a6e6`. The client removed the `releaseFunds()` function.
Furthermore, the client provided the following explanation:

“The owner can steal all the rewards (not just the leftovers) by using withdrawLeftoverRewards()"

- This function is there to prevent reward tokens from being stuck in the contract in cases where a user deploys staking contract for himself, and there not being enough stakers to generate and withdraw all the rewards (a case with 0 stakers is considered possible as well). Additionally, since the existence of reward tokens on the contract directly depends on the owner/deployer of the contract, using this function is akin to the owner not depositing reward tokens at all, which seems impossible to prevent in code. Considering all the above, we’ve decided to leave this function. It’s important to note that the effect of this function can be negated completely by owner renouncing their ownership through the renounceOwnership() function that exists on the contract. This would allow a user (owner) of this contract to adjust this situation based on their case, allowing for more control and withdrawing reward tokens in cases where they were deposited, but nobody actually staked to eventually claim them.

“Can force sending funds back to users by calling releaseFunds()"

- This function is considered redundant and has been removed from the contract in commit (see commit)

“Owner and Operators can start and end a match by calling startMatch() and endMatch(). Note that ZS-5 allows an owner or operator to steal all users' funds in the Escrow contract."

- The nature of the contract and game interaction already assumes a centralized party, which is the game itself. The decisions and data comes from this game and are determined by the game.

> Therefore, a game will have wallets working inside of it that needs special access to this methods to pass the data generated by the game to the contract. In this scenario and at this time there is no way to avoid this. While we are making an initial solution to launch the game with, we will continue working on decentralizing all of our logic further and thinking on ways for the game to be less of a centralized party over time. Meanwhile, we acknowledge the effects of the current pattern and will focus on secure wallet management by the game to avoid compromising our accounts.
>
> “Owner can add and remove operators by calling addOperator() , addOperators() , and removeOperator()" and "Owner can change and revoke ownership through Ownable".
>
> - We consider it a pretty common practice. This is planned and acceptable. Owner is a multisig that can revoke itself if needed, and operators are special access accounts handled by the game to call crucial functions that can not be opened for everyone.
>
> “Owner can set the tokenURI and baseURI by calling setTokenURI() and setBaseURI() respectively"
>
> - Contract belongs to the party that deploys and sets it up. Staked NFT token is created according to the owner of the contract, so any data regarding token URI should be settable by the owner based on the content the owner chose.

Description: In `StakingBase.sol` : `Owner` :

- The owner can steal all the rewards (not just the leftovers) by using `withdrawLeftoverRewards()` .

In `Escrow.sol` : `Owner and Operators` :

- Can force sending funds back to users by calling `releaseFunds()` .

In `Match.sol` : `Owner and Operators` :

- Can start and end a match by calling `startMatch()` and `endMatch()` . Note that ZS-5 allows an owner or operator to steal all users' funds in the Escrow contract.

In `OwnableOperable.sol` : `Owner` :

- Can add and remove operators by calling `addOperator()` , `addOperators()` , and `removeOperator()` .
- Can change and revoke ownership through `Ownable` .

In `StakingERC721.sol` : `Owner` :

- Can set the tokenURI and base tokenURI by calling `setTokenURI` and `setBaseURI` respectively.

Recommendation: This centralization of power needs to be made clear to the users, especially depending on the level of privilege the contract allows to the owner.

Adherence to Best Practices

1. The functions `StakingERC20._mint()` and `StakingERC721._mint()` are not used.
2. In `_checkRewards()` , it doesn't just check for rewards but also updates the unlock timestamp. Consider updating the function naming or doing them in separate functions to have a separation of responsibilities.
3. Unchecked incrementation in the for loop can be removed if using `0.8.22` or above, see the release note for 0.8.22.
4. The function `Match.setFeeVault()` can be replaced by setting `feeVault` to be a public state variable.
5. The use case for `canMatch()` is unclear. The same functionality can be achieved by querying each user's balances off-chain. The client can save contract deployment gas if they remove this function.
6. The `Escrow` contract allows users to deposit and withdraw their funds. However, it also allows an authorized party to return the funds to its rightful owner via the `releaseFunds()` function, which seems unnecessary.
7. In `StakingBase._baseClaim()` , since the transfer would fail if the contract’s balance is less than `rewards` , consider replacing `if (_getContractRewardsBalance() == 0)` with `if (_getContractRewardsBalance() < rewards)` .
8. For consistency and to prevent unexpected behavior in the future, we recommend using a specific up-to-date Solidity version as opposed to an unlocked version (i.e. `^0.8.20`).

Adherence to Specification

1. The NatSpec comments above the `Match.canMatch()` function state that *the returned array will always be the same length as the* `players` *array with valid players being* `address(0)` *in the same index as the player in the* `players` *array*. However, a secondary iteration variable `k` is used to fill the returned array, and it is only increased when the player is not a valid player. If all players are valid except the last one, the returned array would have the invalid player at index 0, and all other valid players’ indexes would not match their index in the `players` array. Evaluate which of the two is correct and adapt the other as required.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Files

- `ed1...4b0 ./contracts/escrow/IEscrow.sol`
- `b59...c92 ./contracts/escrow/Escrow.sol`
- `9cc...c0e ./contracts/staking/StakingBase.sol`
- `75e...094 ./contracts/staking/IStakingBase.sol`
- `3f8...c15 ./contracts/staking/ERC721/StakingERC721.sol`
- `a61...cd5 ./contracts/staking/ERC721/IStakingERC721.sol`
- `ac7...387 ./contracts/staking/ERC20/IStakingERC20.sol`
- `f9d...b32 ./contracts/staking/ERC20/StakingERC20.sol`
- `387...ea9 ./contracts/mock/tokens/MockERC20.sol`
- `a72...b98 ./contracts/mock/tokens/MockERC721.sol`
- `e90...563 ./contracts/match/Match.sol`
- `6e8...3f3 ./contracts/match/IMatch.sol`
- `923...dd0 ./contracts/access/IOwnableOperable.sol`
- `62f...f79 ./contracts/access/OwnableOperable.sol`

Tests

- `354...cc4 ./test/ownable-operable.test.ts`
- `a76...263 ./test/escrow.test.ts`
- `11b...622 ./test/match.test.ts`
- `f08...4ef ./test/staking721.test.ts`
- `273...489 ./test/staking20.test.ts`
- `534...905 ./test/helpers/errors.ts`
- `8eb...521 ./test/helpers/staking/rewards.ts`
- `b63...d96 ./test/helpers/staking/defaults.ts`
- `9ac...b64 ./test/helpers/staking/types.ts`
- `dfd...ab0 ./test/helpers/staking/constants.ts`
- `0d4...f48 ./test/helpers/staking/index.ts`
- `727...d09 ./test/helpers/match/hashing.ts`
- `8da...349 ./test/helpers/match/payouts.ts`
- `5a1...958 ./test/helpers/match/events.ts`

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.10.0

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither .`

Automated Analysis

Slither

All issues reported have either been included in the report or disregarded as false positives.

Test Suite Results

All 134 test cases passed.

```
Escrow Contract
  Deploy
    ✔ Should set the right owner
    ✔ Should set the right token contract
    ✔ Should assign operators in the constructor if they are passed
    ✔ Should revert if _token is passed as a non-contract address (56ms)
  Fund Management - Success Scenarios
    ✔ Should allow deposits (42ms)
    ✔ Should allow to #releaseFunds() by the owner/operator (42ms)
    ✔ Should re-deposit (39ms)
    ✔ Should allow withdrawal (50ms)
    ✔ Should handle withdrawal after multiple deposits (63ms)
    ✔ Should revert on attempt to withdraw with zero balance
    ✔ Should revert on 0 balance to releaseFunds
    ✔ Should revert if depositing zero amount
    ✔ Should revert withdrawals when balance is zero or withdrawing more than balance (62ms)
    ✔ Should handle multiple consecutive deposits and withdrawals correctly (82ms)
    ✔ Should emit the correct events and update balances on multiple #releaseFunds() (97ms)
  Fund Management - Failure Scenarios
    ✔ Should fail an unauthorized #releaseFunds() call
    ✔ Should revert #releaseFunds() called by non-owner/operator
  Deposits of different grade amounts
    ✔ Should handle deposit of 0 wei correctly
    ✔ Should handle deposit of 1 wei correctly
    ✔ Should handle deposit of 100 wei correctly
    ✔ Should handle deposit of 10000000 wei correctly
    ✔ Should handle deposit of 1000000000000000000 wei correctly

  Match Contract
    ✔ Should revert if feeVault is passed as 0x0 address
    Aux Operations
      ✔ #canMatch() should correctly return players with missing funds (141ms)
      ✔ #setFeeVault should set the address correctly and emit an event (66ms)
      ✔ #setFeeVault() should revert if called by non-owner
      ✔ #setFeeVault() should revert if 0x0 address is passed
    Matches
      #startMatch()
        ✔ Should revert if called by a non-owner/non-operator
        ✔ Should fail if any player is not funded
        ✔ Should not start a match with an empty players array
        ✔ Should start match for all funded players and emit MatchStarted event with correct parameters
(173ms)
```

```
        ✔ Should exempt the entry fee from all the player balances
        ✔ Should save and lock the correct amount of fees for the match
        ✔ Should fail if the match already exists
      #endMatch()
        ✔ Should revert if called by a non-owner/non-operator
        ✔ Should fail if the match does not exist
        ✔ Should fail if players and payouts array lengths mismatch
        ✔ Should revert if payout amounts and/or gameFee are calculated incorrectly (83ms)
        ✔ Should end the match and emit event with correct parameters (57ms)
        ✔ Should remove the locked amount from the #lockedFunds mapping
        ✔ Should disperse the payouts correctly to the player #balances
        ✔ Should add #gameFee to the #feeVault balance
        ✔ Players should be able to withdraw their winnings (77ms)
    Access Control
      ✔ owner, operators assigned at deploy and operators assigned later should have appropriate rights
(311ms)

  OwnableOperable Contract
    Ownable
      ✔ should assign msg.sender as owner
      ✔ should #transferOwnership() if owner is the caller
      ✔ should revert when #transferOwnership() is called by non-owner
    Operatable
      ✔ should assign an operator if owner is the caller and emit an event
      ✔ should revert when #addOperator() is called by non-owner
      ✔ should revert when #addOperators() is called by non-owner
      ✔ should revert it zero address is passed as operator
      ✔ should remove an operator if owner is the caller and emit an event
      ✔ should revert when #removeOperator() is called by non-owner
      ✔ should support multiple operators (55ms)

  StakingERC20
    #getContractRewardsBalance
      ✔ Allows a user to see the total rewards remaining in a pool
    #stake
      ✔ Can stake an amount successfully (51ms)
      ✔ Can stake a second time as the same user successfully (54ms)
      ✔ Can stake as a new user when others are already staked (43ms)
      ✔ Fails when the staker doesn't have the funds to stake (78ms)
      ✔ Fails when the staker tries to stake 0
    #getRemainingLockTime
      ✔ Allows the user to view the remaining time lock period for a stake
      ✔ Returns 0 for a user that's passed their lock time
      ✔ Returns 0 for a user that has not staked
    #getPendingRewards
      ✔ Allows the user to view the pending rewards for a stake
      ✔ Returns 0 for a user that has not staked
      ✔ Returns 0 for a user that has staked but not passed a time period
    #claim
      ✔ Allows the user to claim their rewards
      ✔ Fails when the user has never staked
      ✔ Fails when the contract has no rewards
      ✔ Fails when the user has not passed their lock time
    #unstake
      ✔ Allows a user to unstake partially (57ms)
      ✔ Allows a user to fully withdraw their entire staked amount (51ms)
      ✔ Fails when the user has never staked
      ✔ Fails when the user has not passed their lock time
      ✔ Fails when the user tries to unstake more than they have staked
    #unstake with 'exit'
      ✔ Allows a user to partially unstake without rewards using 'exit' (40ms)
      ✔ Allows a user to fully unstake without rewards using 'exit' (39ms)
      ✔ Fails when the user has never staked
      ✔ Succeeds when the user has not passed their lock time
      ✔ Fails when the user tries to unstake more than they have staked
    #withdrawLeftoverRewards
      ✔ Allows the admin to withdraw leftover rewards
      ✔ Fails when the caller is not the admin
      ✔ Fails when the contract has no rewards left to withdraw
    Events
```

```
      ✔ Emits a Staked event when a user stakes
      ✔ Emits a Claimed event when a user claims rewards (47ms)
      ✔ Emits an Unstaked event when a user unstakes (61ms)
      ✔ Emits an Unstaked event when a user exits with unstake (60ms)
      ✔ Emits 'LeftoverRewardsWithdrawn' event when the admin withdraws

  StakingERC721
    ✔ Should NOT deploy with zero values passed (148ms)
    #getContractRewardsBalance
      ✔ Allows a user to see the total rewards remaining in a pool
    #stake
      ✔ Can stake an NFT and properly assign tokenURI using baseURI (53ms)
      ✔ Can stake multiple NFTs (63ms)
      ✔ Fails when the user tries to transfer the SNFT
      ✔ Fails to stake when the token id is invalid (38ms)
      ✔ Fails to stake when the token is already staked (40ms)
      ✔ Fails to stake when the caller is not the owner of the NFT
    #getRemainingLockTime
      ✔ Allows the user to view the remaining time lock period for a stake
      ✔ Returns 0 for a staked user that is past their lock time
      ✔ Returns 0 for users that have not staked
    #getPendingRewards
      ✔ Can view pending rewards for a user
      ✔ Returns 0 for users that have not passed a single time period
      ✔ Returns 0 for users that have not staked
    #claim
      ✔ Can claim rewards when staked and past the timeLockPeriod (54ms)
      ✔ Fails to claim when not enough time has passed (44ms)
      ✔ Fails to claim when the caller has no stakes
    #unstake
      ✔ Can unstake a token (65ms)
      ✔ Can unstake multiple staked tokens (85ms)
      ✔ Fails to unstake when not enough time has passed
      ✔ Fails to unstake when token id is invalid
      ✔ Fails to unstake when caller is not the owner of the SNFT (64ms)
      ✔ Fails to unstake when token id is not staked
    #unstake with 'exit'
      ✔ Fails if the caller does not own the sNFT
      ✔ Fails if the sNFT is invalid
      ✔ Allows the user to remove their stake within the timelock period without rewards (58ms)
      ✔ Allows the user to remove multiple stakes within the timelock period without rewards (103ms)
    Events
      ✔ Staking emits a 'Staked' event (52ms)
      ✔ Staking multiple tokens emits multiple 'Staked' events (109ms)
      ✔ Claim emits a 'Claimed' event (40ms)
      ✔ Unstake Emits 'Unstaked' and 'Claimed 'events (80ms)
      ✔ Unstaking multiple tokens emits multiple 'Unstaked' and 'Claimed' events (128ms)
    Other configs
      ✔ Can't use the StakingERC721 contract when an IERC20 is the staking token (79ms)
      ✔ Can't use the StakingERC721 contract when an IERC721 is the rewards token (201ms)
      ✔ Can't use 0 as the period length (53ms)
      ✔ Can't transfer rewards when there is no rewards balance (112ms)
      ✔ More complex use case involving multiple stakes and stakers (1262ms)
    Helper functions
      ✔ #setBaseURI() should set the base URI (43ms)
      ✔ #setTokenURI() should set the token URI and return it properly when baseURI is empty (43ms)
      ✔ #stake() with passed tokenURI should set the token URI when baseURI is empty and change back to
  baseURI when needed (69ms)
      ✔ #withdrawLeftoverRewards() should withdraw all remaining rewards and emit an event
      ✔ #withdrawLeftoverRewards() should revert if contract balance is 0
      ✔ #withdrawLeftoverRewards() should only be callable by the owner
      ✔ #supportsInterface() should return true for ERC721 interface or interface of staking contract
      ✔ Should allow to change ownership
      ✔ should allow to renounce ownership


  134 passing (8s)
```

Code Coverage

The code coverage is acceptable, with an average of 79% branch coverage. We recommend the client to improve upon the branch coverage to be above 90% and add additional tests to validate fixes for the issues identified in the audit.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
access/	100	100	100	100	
IOwnableOperable.sol	100	100	100	100	
OwnableOperable.sol	100	100	100	100	
escrow/	100	85.71	100	100	
Escrow.sol	100	85.71	100	100	
IEscrow.sol	100	100	100	100	
match/	100	100	100	100	
IMatch.sol	100	100	100	100	
Match.sol	100	100	100	100	
staking/	86.36	63.64	72.73	87.88	
IStakingBase.sol	100	100	100	100	
StakingBase.sol	86.36	63.64	72.73	87.88	169,178,183,184
staking/ERC20/	71.43	70	100	66.67	
IStakingERC20.sol	100	100	100	100	
StakingERC20.sol	71.43	70	100	66.67	... 87,88,92,94
staking/ERC721/	75.68	60	82.35	77.78	
IStakingERC721.sol	100	100	100	100	
StakingERC721.sol	75.68	60	82.35	77.78	... 248,249,252
All files	86.55	79	87.76	87.08	

Changelog

- 2024-05-24 - Initial report
- 2024-06-13 - Final report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites&aspo; owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that your access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.