SeaPad Smart Contract Audit Report

MOVEBIT

contact@movebit.xyz

https://twitter.com/movebit_

07/24/2023

SeaPad Smart Contract Audit Report

1 Executive Summary

1.1 Project Information

Description	A decentralized launchpad on Sui
Type	Launchpad
Auditors	MoveBit
Timeline	July 6, 2023 – July 21, 2023
Languages	Move
Platform	Sui
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/seapad-fund/sui-contracts/tree/mainnet/vesting/sources
Commits	4d13c702821251230ae115ce6229d5904ec1cfdd d4a37da96f7a9d468538ae8fddc3762421988f08

1.2 Files in Scope

The following are the SHA1 hashes of the last reviewed files.

ID	Files	SHA-1 Hash
TML	sui-contracts/vesting/Move.toml	e0c0c836544fb2a3d2d9928cc6b9fef346499fb5

VER	sui-contracts/vesting/sources/version.move	d5c31f42f8d5338327c275572f51edc3166dc518
VES	sui-contracts/vesting/sources/vesting.move	dea615ba5e7e504d1fe9856f1e412a3bef7357a7

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	11	10	1
Informational	2	2	
Minor	2	2	
Medium	3	3	
Major	4	3	1
Critical			

1.4 MoveBit Audit BreakDown

MoveBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow by bit operations
- Number of rounding errors
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification

- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting
- Unchecked CALL Return Values
- The flow of capability
- Witness Type

1.5 Methodology

The security team adopted the "**Testing and Automated Analysis**", "**Code Review**" and "**Formal Verification**" strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section **1.2**.

(3) Formal Verification

Perform formal verification for key functions with the Move Prover.

(4) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by **Seapad** to identify any potential issues and vulnerabilities in the source code of the **Seapad Vesting** smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we have identified **11** issues of varying severity, listed below.

ID	Title	Severity	Status
VES-01	Missing Deprecated Check During Fund Addition	Major	Fixed
VES-02	Single-step Ownership Transfer can be Dangerous	Medium	Fixed
VES-03	The Value of `token_fund.percent` is not Updated	Medium	Fixed
VES-04	Assertion is Unnecessary	Minor	Fixed
VES-05	Consolidating Redundant Table Access	Minor	Fixed
VES-06	Unremoved Entries After Claiming All Locked Tokens	Major	Fixed
VES-07	The `project.deposited` is not Updated When Claiming Tokens	Major	Fixed
VES-08	Possible Zero Percent in `addFund()` Function	Medium	Fixed
VES-09	Centralization Risk	Major	Acknowledged
VES-10	Unused Error Code `ERR_CONFIRMED_ADMINCAP`	Informational	Fixed
VES-11	`value_fund > 0` Check is Duplicated in `addFunds`	Informational	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the `SeaPad Vesting` Smart Contract:

Admin

- Admin can change Admin through `changeAdmin`
- Admin can create new projects through `createProject`
- Admin can set if the project is deprecated by `setDeprecated`
- Admin can set the project fee by `setProjectFee`
- Admin can withdraw all the project fee by `withdrawFee`
- Admin can add a single fund by `addFund`
- Admin can add multiple funds by `addFunds`
- Admin can remove fund by `removeFund`

User

- User can claim their locked funds from a project by `claim`

4 Findings

VES-01 Missing Deprecated Check During Fund Addition in the Protocol

Severity: Major

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L221-L240

Descriptions:

This function, `setDeprecated()` , is a public entry function that allows the administrator, represented by the _admin parameter, to set the deprecation status of a Project object.

```
public entry fun changeAdmin(admin: VAdminCap, to: address, version: &mut Version) {
    checkVersion(version, VERSION);
    transfer(admin, to);
}

```

In the protocol, when adding funds, there is no check performed on the deprecated status of the project. This means that funds can be added to a project without considering whether the project has been marked as deprecated or not. This can lead to potential issues, as funds may be inadvertently added to projects that are no longer actively supported or recommended.

```
public entry fun addFunds<COIN>(admin: &VAdminCap,
    owners: vector<address>,
    values: vector<u64>,
    totalFund: Coin<COIN>,
    project: &mut Project<COIN>,
    registry: &mut ProjectRegistry,
    version: &Version,
    ctx: &mut TxContext) {
        let (i, n) = (0, vector::length(&owners));
        assert!(vector::length(&values) == n, ERR_BAD_FUND_PARAMS);
        while (i < n) {
            let owner = *vector::borrow(&owners, i);
            let value_fund = *vector::borrow(&values, i);
            assert!(value_fund > 0, ERR_BAD_FUND_PARAMS);
            let fund = coin::split(&mut totalFund, value_fund, ctx);
            addFund(admin, owner, fund, project, registry, version);
            i = i + 1;
        };
        transfer::public_transfer(totalFund, sender(ctx));
    }

```

Suggestion: It is important to implement proper checks and validations during the fund addition process to ensure that deprecated projects are not able to receive new funds. This helps maintain consistency and aligns with the intended deprecation status of the projects in the protocol.

Resolution: Added the code `assert!(!project.deprecated, ERR_BDEPRECATED)` to check if the project has been deprecated.

VES–02 Single–step Ownership Transfer Can be Dangerous

Severity: Medium

Status: Fixed

Code Location: sui–contracts/vesting/sources/version.move#L117–L120

Descriptions:

Single–step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever. If the admin permissions are given to the wrong address within this function, it will cause irreparable damage to the contract.

```
public entry fun changeAdmin(admin: VAdminCap, to: address, version: &mut Version) {
    checkVersion(version, VERSION);
    transfer(admin, to);
}

```

Suggestion: It is a best practice to use a two–step ownership transfer pattern, meaning ownership transfer gets to a "pending" state and the new owner should claim his new rights, otherwise the old owner still has control of the contract.

Resolution: Two–step ownership transfer applied.

VES–03 The Value of `token_fund.percent` is not Updated

Severity: Medium

Status: Fixed

Code Location: sui–contracts/vesting/sources/version.move#L328–L367

Descriptions:

When funds are added to an existing owner's entry in the `project.funds` table, the `token_fund.percent` is incremented by the percentage of the added funds relative to the project's total supply. However, there is no corresponding logic to update or adjust this value during the claim process.

This means that if funds are added multiple times for the same owner, the token_fund.percent will accumulate the percentages of all the added funds without considering any changes or claims made on the funds. As a result, the `token_fund.percent` value will be inaccurate and may not reflect the actual percentage of funds owned by the owner.

```
public entry fun claim<COIN>(fee: &mut Coin<SUI>,
project: &mut Project<COIN>,
sclock: &Clock,
version: &Version,
ctx: &mut TxContext) {
    checkVersion(version, VERSION);
    assert!(coin::value(fee) >= project.fee, ERR_FEE_NOT_ENOUGH);
    let now_ms = clock::timestamp_ms(sclock);
    assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED);
    let sender_addr = sender(ctx);
    assert!(table::contains(&project.funds, sender_addr), ERR_NO_FUND);
    assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED);
    let fund0 = table::borrow(&mut project.funds, sender_addr);
    assert!(sender_addr == fund0.owner, ERR_NO_PERMISSION);
    let claim_percent = computeClaimPercent<COIN>(project, now_ms);
    assert!(claim_percent > 0, ERR_NO_FUND);
    let fund = table::borrow_mut(&mut project.funds, sender_addr);
    let claim_total = (fund.total * claim_percent) / ONE_HUNDRED_PERCENT_SCALED;
    let claim = claim_total - fund.released;
    assert!(claim > 0, ERR_NO_FUND);
    transfer::public_transfer(coin::split<COIN>(&mut fund.locked, claim, ctx), sender_addr);
    fund.released = fund.released + claim;
    fund.last_claim_ms = now_ms;
    coin::join(&mut project.feeTreasury, coin::split(fee, project.fee, ctx));
    emit(FundClaimEvent {
        owner: fund.owner,
        total: fund.total,
        released: fund.released,
        claim,
        project: id_address(project),
    })
}
```

Suggestion: Appropriate logic should be implemented to update the token_fund.percent value during the claim process or any other relevant operations to reflect the correct percentage of funds owned by the owner. This ensures that the token_fund.percent remains accurate and aligned with the actual ownership of funds within the project.

Resolution: Remove the `token_fund.percent` feature.

VES-04 The Assertion `assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED)` in the Code is Unnecessary

Severity: Minor

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L341

Descriptions:

The line 14 `assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED)` in the code below is redundant.

The purpose of this assertion is to verify that the current timestamp (now_ms) is greater than or equal to the TGE (Token Generation Event) start time of the project (project.tge_ms). However, this check is already performed earlier in the code, right after retrieving the current timestamp. Therefore, this second assertion serves no additional purpose and can be safely removed without affecting the functionality of the function.

```
1   public entry fun claim<COIN>(fee: &mut Coin<SUI>,
2       project: &mut Project<COIN>,
3       sclock: &Clock,
4       version: &Version,
5       ctx: &mut TxContext) {
6       checkVersion(version, VERSION);
7       assert!(coin::value(fee) >= project.fee, ERR_FEE_NOT_ENOUGH);
8       let now_ms = clock::timestamp_ms(sclock);
9       assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED);
10      let sender_addr = sender(ctx);
11      assert!(table::contains(&project.funds, sender_addr), ERR_NO_FUND);
12      assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED);
13      let fund0 = table::borrow(&mut project.funds, sender_addr);
14      assert!(sender_addr == fund0.owner, ERR_NO_PERMISSION);
15      let claim_percent = computeClaimPercent<COIN>(project, now_ms);
16      assert!(claim_percent > 0, ERR_NO_FUND);
17      let fund = table::borrow_mut(&mut project.funds, sender_addr);
18      let claim_total = (fund.total * claim_percent) / ONE_HUNDRED_PERCENT_SCALED;
19      let claim = claim_total - fund.released;
20      assert!(claim > 0, ERR_NO_FUND);
21      transfer::public_transfer(coin::split<COIN>(&mut fund.locked, claim, ctx), sender_addr);
22      fund.released = fund.released + claim;
23      fund.last_claim_ms = now_ms;
24      coin::join(&mut project.feeTreasury, coin::split(fee, project.fee, ctx));
25      emit(FundClaimEvent {
26          owner: fund.owner,
27          total: fund.total,
28          released: fund.released,
29          claim,
30          project: id_address(project),
31      })
32      }
```

Suggestion: Remove the line 14 `assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED)` from the code.

Resolution: Duplicate check removed.

VES-05 Consolidating Redundant Table Access

Severity: Minor

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L343-L349

Descriptions: The code could be optimized by merging the two references to `table::borrow(&mut project.funds, sender_addr)` into a single occurrence. Currently, the code makes two separate calls to retrieve the same value from the `project.funds` table, which is inefficient. By consolidating these references into a single call, the code can improve performance and reduce redundant code execution.

```
public entry fun claim<COIN>(fee: &mut Coin<SUI>,
project: &mut Project<COIN>,
sclock: &Clock,
version: &Version,
ctx: &mut TxContext) {
    checkVersion(version, VERSION);
    assert!(coin::value(fee) >= project.fee, ERR_FEE_NOT_ENOUGH);
    let now_ms = clock::timestamp_ms(sclock);
    assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED);
    let sender_addr = sender(ctx);
    assert!(table::contains(&project.funds, sender_addr), ERR_NO_FUND);
    assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED);
    let fund0 = table::borrow(&mut project.funds, sender_addr);
    assert!(sender_addr == fund0.owner, ERR_NO_PERMISSION);
    let claim_percent = computeClaimPercent<COIN>(project, now_ms);
    assert!(claim_percent > 0, ERR_NO_FUND);
    let fund = table::borrow_mut(&mut project.funds, sender_addr);
    let claim_total = (fund.total * claim_percent) / ONE_HUNDRED_PERCENT_SCALED;
    let claim = claim_total - fund.released;
    assert!(claim > 0, ERR_NO_FUND);
    transfer::public_transfer(coin::split<COIN>(&mut fund.locked, claim, ctx), sender_addr);
    fund.released = fund.released + claim;
    fund.last_claim_ms = now_ms;
    coin::join(&mut project.feeTreasury, coin::split(fee, project.fee, ctx));
    emit(FundClaimEvent {
        owner: fund.owner,
        total: fund.total,
        released: fund.released,
        claim,
        project: id_address(project),
    })
}

```

Suggestion: It is recommended to modify it like this

```
let claim_percent = computeClaimPercent<COIN>(project, now_ms);
assert!(claim_percent > 0, ERR_NO_FUND);

let fund = table::borrow_mut(&mut project.funds, sender_addr);
assert!(sender_addr == fund.owner, ERR_NO_PERMISSION);
```

Resolution: The developers have fixed this issue based on our recommendation.

VES-06 Unremoved Entries After Claiming All Locked Tokens

Severity: Major

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L384-L425

Descriptions: The `claim()` function is used to claim a certain amount of tokens in a project. If user claims all of the locked amount but the protocol does not remove the user's entry from the table with `table::remove(&mut registry.user_projects, owner)` , it could lead to potential issues.

One such issue is that the user's entry would still exist in the table, even though they have claimed all their tokens. This could lead to confusion or incorrect assumptions when querying the table for data, as it might appear that the user still has tokens to claim, even though they do not.

```
public entry fun claim<COIN>(fee: Coin<SUI>,
    project: &mut Project<COIN>,
    sclock: &Clock,
    version: &Version,
    ctx: &mut TxContext) {
        checkVersion(version, VERSION);
        assert!(coin::value(&fee) >= project.fee, ERR_FEE_NOT_ENOUGH);
    let now_ms = clock::timestamp_ms(sclock);
    assert!(now_ms >= project.tge_ms, ERR_TGE_NOT_STARTED);

    let sender_addr = sender(ctx);
    assert!(table::contains(&project.funds, sender_addr), ERR_NO_FUND);

    let claim_percent = computeClaimPercent<COIN>(project, now_ms);
    assert!(claim_percent > 0, ERR_NO_FUND);

    let token_fund = table::borrow_mut(&mut project.funds, sender_addr);
    assert!(sender_addr == token_fund.owner, ERR_NO_PERMISSION);

    let claim_total = (token_fund.total * claim_percent) / ONE_HUNDRED_PERCENT_SCALED;
    let claimed_amount = claim_total - token_fund.released;
    assert!(claimed_amount > 0, ERR_NO_FUND);

    let percent = claimed_amount * ONE_HUNDRED_PERCENT_SCALED / project.supply;
    token_fund.percent = token_fund.percent - percent;

    transfer::public_transfer(coin::split<COIN>(&mut token_fund.locked, claimed_amount, ctx), sender_addr);
    token_fund.released = token_fund.released + claimed_amount;
    token_fund.last_claim_ms = now_ms;

    let takeFee = coin::split(&mut fee, project.fee, ctx);
    coin::join(&mut project.feeTreasury, takeFee);
    transfer::public_transfer(fee, sender(ctx));

    emit(FundClaimEvent {
        owner: token_fund.owner,
        total: token_fund.total,
        released: token_fund.released,
        claim: claimed_amount,
        project: id_address(project),
    })
}
```

Suggestion: It would be advisable to add a check after the tokens are claimed and if the locked amount is zero, then remove the user's entry from the table.

Resolution: Clear user from table if user claim all token.

VES-07 The `project.deposited` is not Updated When Claiming Tokens

Severity: Major

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L377-L432

Descriptions: This function does not update `project.deposited` when a user claims their tokens. if admin removes funds with a `removeFund` operation and `project.deposited_percent` is calculated based on the `project.deposited` , it leads to incorrect calculations because `project.deposited` isn't being updated correctly in the claim function.

```
let claim_total = (token_fund.total * claim_percent) / ONE_HUNDRED_PERCENT_SCALED;
let claimed_amount = claim_total - token_fund.released;
assert!(claimed_amount > 0, ERR_NO_FUND);

let percent = claimed_amount * ONE_HUNDRED_PERCENT_SCALED / project.supply;
token_fund.percent = token_fund.percent - percent;

transfer::public_transfer(coin::split<COIN>(&mut token_fund.locked, claimed_amount, ctx), sender_addr);
token_fund.released = token_fund.released + claimed_amount;
token_fund.last_claim_ms = now_ms;

let takeFee = coin::split(&mut fee, project.fee, ctx);
coin::join(&mut project.feeTreasury, takeFee);
transfer::public_transfer(fee, sender(ctx));
```

Suggestion: Update the `project.deposited` value after claiming tokens

Resolution: `project.deposited` has been updated.

VES-08 Possible Zero Percent in `addFund()` Function

Severity: Medium

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L315

Descriptions: In the function `addFund()` , there's a line of code that calculates the percent of the total supply that the new funds (`fund_amt`) represent:

```
let percent = fund_amt * ONE_HUNDRED_PERCENT_SCALED / project.supply;

```

This formula takes the amount of funds being added, scales it up by ONE_HUNDRED_PERCENT_SCALED (likely a large constant for scaling purposes), and then divides it by the total supply of the project.

The issue you're referring to arises when the `fund_amt` is very small relative to `project.supply` . In such cases, when `fund_amt` * `ONE_HUNDRED_PERCENT_SCALED` is divided by `project.supply` , the result may be rounded down to zero due to the way integer division works in many programming languages. This can happen even if `fund_amt` is not exactly zero, just very small compared to `project.supply` .

Suggestion: Assert when percent is 0:

`assert!(percent > 0, "Zero percent");`

Resolution: Removed the `token_fund.percent` feature.

VES-09 Centralization Risk

Severity: Major

Status: Acknowledged

Descriptions: There are some centralization risks in the contract:

- Admin can change Admin through `changeAdmin`
- Admin can create new projects through `createProject`
- Admin can set if the project is deprecated by `setDeprecated`
- Admin can set the project fee by `setProjectFee`
- Admin can withdraw all the project fee by `withdrawFee`
- Admin can add a single fund by `addFund`

- Admin can add multiple funds by `addFunds`
- Admin can remove fund by `removeFund`

Suggestion: It is recommended that multi-signature accounts should be set as privileged accounts.

VES-10 Unused Error Code `ERR_CONFIRMED_ADMINCAP`

Severity: Informational

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L38

Descriptions: Error code `ERR_CONFIRMED_ADMINCAP` is not used anywhere.

```
const ERR_CONFIRMED_ADMINCAP: u64 = 8010;
```

Suggestion: Consider removing it if it's unused.

Resolution: Removed useless code.

VES-11 `value_fund > 0` Check is Duplicated in `addFunds`

Severity: Informational

Status: Fixed

Code Location: sui-contracts/vesting/sources/version.move#L286-307

Descriptions: In the `addFunds` function, there is a check to make sure `value_fund` is greater than 0 in the while loop. We think it's unnecessary since in each iteration it will call `addFund` function, and it already contains the same check.

```
assert!(value_fund > 0, ERR_BAD_FUND_PARAMS);
let fund = coin::split(&mut totalFund, value_fund, ctx);
addFund(admin, owner, fund, project, registry, version);
```

```
public entry fun addFund<COIN>(_admin: &AdminCap,
                                owner: address,
                                fund: Coin<COIN>,
                                project: &mut Project<COIN>,
                                registry: &mut ProjectRegistry,
                                version: &Version)
{
    checkVersion(version, VERSION);

    assert!(!project.deprecated, ERR_BDEPRECATED);

    let fund_amt = coin::value(&fund);
    assert!(fund_amt > 0, ERR_BAD_FUND_PARAMS);
```

Suggestion: Consider removing the duplicate test in `addFunds` .

Resolution: Removed the duplicate check.

Appendix 1

Issue Level

- **Informational:** Informational items are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.

- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.