VTVL Vesting

Security Assessment

March 21st, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Woosun Song procfs@osec.io

Table of Contents

Appendices

© 2024 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

VTVL engaged OtterSec to assess the `vesting` programs. This assessment was conducted between March 5th and March 11th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we noticed that the mint limit of `VariableSupplyERC20Token` may be bypassed to mint beyond the initially set bounds (OS-VTVL-ADV-00).

We also made recommendations around potential asset freezing hazards in claim revocation (OS-VTVL-SUG-00) which was addressed through off-chain mechanics. Furthermore, we provided a list of suggestions to improve code maturity and optimize gas consumption (OS-VTVL-SUG-01).

© 2024 Otter Audits LLC. All Rights Reserved.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/VTVL-co/vtvl-smart-ontracts/. This audit was performed against commit `6a5779` .

A brief description of the programs is as follows:

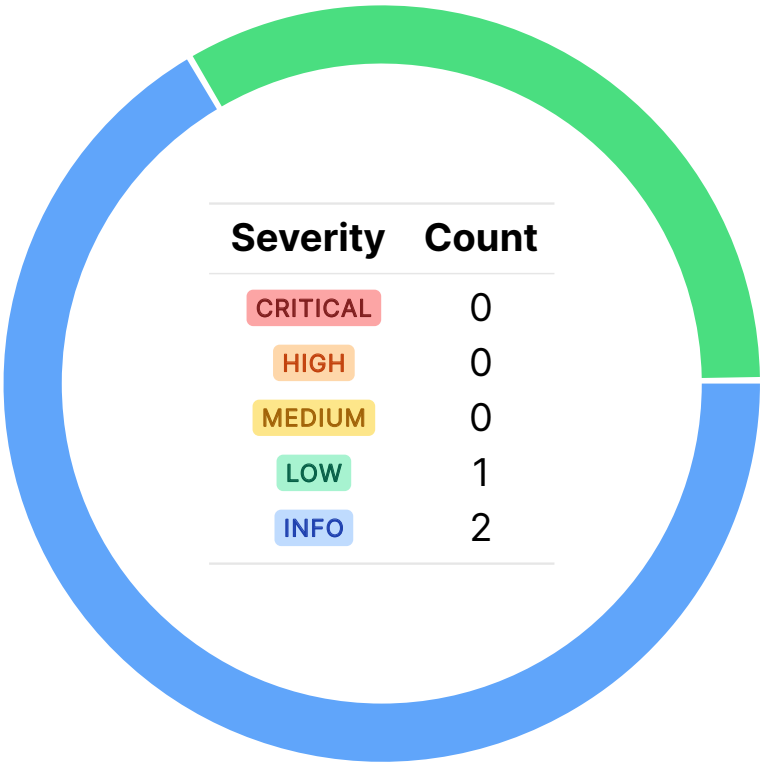
Name	Description
vesting-contracts	These contracts serve as the primary components governing the vesting process of a particular token. They support two types of vesting mechanisms: cliff and linear. With cliff vesting, the entire allocation is released at once, specified by a particular timestamp, while linear vesting distributes the allocation over a defined period, gradually increasing its release.

© 2024 Otter Audits LLC. All Rights Reserved.

03 — Findings

Overall, we reported 3 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 do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	1
INFO	2

© 2024 Otter Audits LLC. All Rights Reserved.

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 A.

ID	Severity	Status	Description
OS-VTVL-ADV-00	LOW	TODO	The mint limit in `VariableSupplyERC20Token` may be bypassed.

© 2024 Otter Audits LLC. All Rights Reserved.

Mint Limit Bypass LOW

OS-VTVL-ADV-00

Description

The mint limit in `VariableSupplyERC20Token` may be bypassed. This bypass is feasible because it uses the value zero to denote unlimited cap. Therefore, once the entire mint limit is consumed, the limit is perpetually relaxed.

>_ *contracts/token/VariableSupplyERC20Token.sol* solidity

```solidity
function mint(address account, uint256 amount) public onlyAdmin {
    require(account != address(0), "INVALID_ADDRESS");
    // If we're using maxSupply, we need to make sure we respect it
    // mintableSupply = 0 means mint at will
    if (mintableSupply > 0) {
        require(amount <= mintableSupply, "INVALID_AMOUNT");
        // We need to reduce the amount only if we're using the limit, if not just leave it be
        mintableSupply -= amount;
    }
    _mint(account, amount);
}
```

In its `mint` function, the mint limit checks are only performed if `mintableSupply` is nonzero. Assuming an initial `mintableSupply` of `1e18`, an attacker with admin roles may bypass this limit by minting precisely `1e18` tokens, reducing `mintableSupply` to zero. Afterwards, it can mint an unlimited amount of tokens because the branch is not taken afterwards.

Remediation

Use `type(uint256).max` to denote unlimited mint limit and remove the nonzero check.

solidity

```solidity
function mint(address account, uint256 amount) public onlyAdmin {
    require(account != address(0), "INVALID_ADDRESS");
    require(amount <= mintableSupply, "INVALID_AMOUNT");
    mintableSupply -= amount;
    _mint(account, amount);
}
```

Patch

Fixed in d41cedc.

© 2024 Otter Audits LLC. All Rights Reserved.

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 anti-patterns and may result in security issues in the future.

ID	Description
OS-VTVL-SUG-00	`revokeClaim` causes the so-far vested amount to be unwithdrawable.
OS-VTVL-SUG-01	We made recommendations around code maturity and gas optimization.

© 2024 Otter Audits LLC. All Rights Reserved.

Frozen Funds

OS-VTVL-SUG-00

Description

`revokeClaim` causes the so-far vested amount to be unwithdrawable. This occurs because withdrawals are only allowed for claims with the `isActive` flag. Thus, the `vestedSoFarAmt` number of tokens are not claimable by the vest recipient. Furthermore, it is not withdrawable by the admin either, because only `amountRemaining` is deducted from `numTokensReservedForVesting`.

>_ *contracts/VTVLVesting.sol* solidity

```solidity
function withdraw() external hasActiveClaim(_msgSender()) nonReentrant {
    /* ... */
}

function revokeClaim(address _recipient) external onlyAdmin
    hasActiveClaim(_recipient)
{
   /* ... */
    _claim.isActive = false;
    _claim.deactivationTimestamp = uint40(block.timestamp);

    // The amount that is "reclaimed" is equal to the total allocation less what was already
        ↪  withdrawn
    uint256 vestedSoFarAmt = vestedAmount(_recipient, uint40(block.timestamp));
    uint256 amountRemaining = finalVestAmt - vestedSoFarAmt;
    numTokensReservedForVesting -= amountRemaining;
}
```

Consequently, revoking a vesting schedule causes the so-far vested amount to be permanently locked in the contract.

Remediation

The VTVL team decided to inform vest owners that they should notify the vest recipients before any claims are revoked, ensuring that the amount vested thus far is depleted upon revocation.

© 2024 Otter Audits LLC. All Rights Reserved.

Code Maturity Recommendations OS-VTVL-SUG-01

Description

1. The for-loop in `createClaimsBatch` may be gas-optimized by changing `i++` to an unchecked

>_ *contracts/VTVLVesting.sol* solidity

```
function createClaimsBatch(
    /* ... */
) external onlyAdmin {
   /* ... */
    for (uint256 i = 0; i < length; i++) {
        /* ... */
    }
}
```

2. In `withdrawOtherToken`, an explicit balance check is unncessary as identical checks are performed within `ERC20.transfer`.

>_ *contracts/VTVLVesting.sol* solidity

```
function withdrawOtherToken(IERC20 _otherTokenAddress)
    external
    onlyAdmin
    nonReentrant
{
    /* ... */
    uint256 bal = _otherTokenAddress.balanceOf(address(this));
    require(bal > 0, "INSUFFICIENT_BALANCE");
    _otherTokenAddress.safeTransfer(_msgSender(), bal);
}
```

3. The `finalClaimableAmount` function may be optimized by changing the location of the `_claim` variable from `storage` to `memory`. This is because using a storage location causes two storage reads, first when reading the individual fields and second when reading the entirety of `_claim` during the invocation of `_baseVestedAmount`. On the other hand, using a memory location reduces the number of storage reads to one, because it only occurs when reading to the `_claim` variable.

>_ *contracts/VTVLVesting.sol* solidity

```
function finalClaimableAmount(address _recipient) external view returns (uint256) {
    Claim storage _claim = claims[_recipient];
    uint40 vestEndTimestamp = _claim.isActive ? _claim.endTimestamp :
        ↪  _claim.deactivationTimestamp;
```

© 2024 Otter Audits LLC. All Rights Reserved.

```
    return _baseVestedAmount(_claim, vestEndTimestamp) - _claim.amountWithdrawn;
}
```

4. The usage of `msg.sender` in `FullPremintERC20Token` should be replaced with `_msgSender()` for coherence.

>_ *contracts/token/FullPremintERC20Token.sol* solidity

```solidity
contract FullPremintERC20Token {
    constructor(
        string memory name_,
        string memory symbol_,
        uint256 supply_,
        bool burnable_
    ) ERC20(name_, symbol_) {
        require(supply_ > 0, "NO_ZERO_MINT");
        _mint(_msgSender(), supply_);
        deployer = msg.sender;
        burnable = burnable_;
    }
}
```

Patch

1. 707568d.
2. c1da74f.
3. 4ebfefc
4. eb5fbda

© 2024 Otter Audits LLC. All Rights Reserved.

A — Vulnerability Rating Scale

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

CRITICAL

Vulnerabilities that immediately result in a 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 may result in a 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 may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

© 2024 Otter Audits LLC. All Rights Reserved.

B — 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 program's implementation 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 thumb, 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 comprehensively understand 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.

© 2024 Otter Audits LLC. All Rights Reserved.