VTVL

Audit

Presented by:

OtterSec contact@osec.io

Ajay Kunapareddy d1r3wolf@osec.io

Robert Chen r@osec.io

Contents

01 | Executive Summary

Overview

VTVL engaged OtterSec to perform an assessment of the `vesting` program. This assessment was conducted between July 24th and July 26th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 19 findings in total.

In particular, we discovered a critical issue related to the improper calculation of fees, resulting in attempts to deduct significantly high fees from users' accounts (OS-VTVL-ADV-00). Furthermore, we identified an issue in which the token address arguments passed to the function for retrieving quote prices become swapped, resulting in incorrect quote amounts (OS-VTVL-ADV-01), and also an instance where the vested amount was incorrectly derived due the division rounding down, resulting in a portion of the amount being locked (OS-VTVL-ADV-07).

We also made recommendations around gas optimization with regard to redundant code (OS-VTVL-SUG-02) and advised utilizing the `ReentrancyGuard` modifier for certain functions (OS-VTVL-SUG-03). We also suggested the utilization of bounds while setting the fee percentage to limit the range of values it may take to avoid extremely low or high fee values (OS-VTVL-SUG-06).

02 | Scope

The source code was delivered to us in a git repository at github.com/VTVL-co/vtvl-smart-contracts/tree/audit-ready-jul-23. This audit was performed against commit 3de0c6c. In addition, we performed supplementary reviews up to commit ba635ec

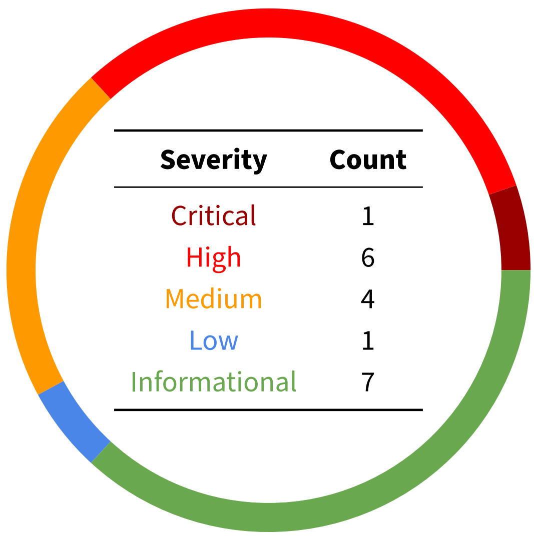
A brief description of the programs is as follows.

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.
milestone-contracts	Implements functionality related to managing milestone-based vesting of tokens. It allows the allocation of tokens to multiple recipients based on specific milestone periods and release intervals.
token-contract	An `ERC20` token contract with unrestricted minting capabilities, providing the option to have either a limited or unlimited supply while prohibiting token burning.

03 | Findings

Overall, we reported 19 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 help mitigate future vulnerabilities.



Severity	Count
Critical	1
High	6
Medium	4
Low	1
Informational	7

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	Critical	Resolved	The conversion of `_realFeeAmount` to USDC is mishandled, resulting in incorrect fee amounts. Additionally, the utilization of `conversionThreshold` results in instances where the correct fee amount is not achieved.
OS-VTVL-ADV-01	High	Resolved	Swapping of token address arguments passed to `getQuoteAtTick` results in incorrect token prices.
OS-VTVL-ADV-02	High	Resolved	An invalid check within `mint` allows the minting of tokens above the maximum allowed limit, effectively providing unlimited minting capability even when the intent is to limit the supply of tokens.
OS-VTVL-ADV-03	High	Resolved	Lack of check on the initial amount of tokens minted results in the initial supply exceeding the maximum stipulated amount of tokens mintable.
OS-VTVL-ADV-04	High	Resolved	The formula that calculates the quoted price underflows in specific instances.
OS-VTVL-ADV-05	High	Resolved	Inaccurate check to determine if the contract has received enough tokens to cover all milestones for all recipients, results in the permanent locking of tokens.
OS-VTVL-ADV-06	High	Resolved	The calculation of `availableAmount` in `withdrawAdmin` locks tokens, showing a false value of the withdrawable balance by the admin.

OS-VTVL-ADV-07	Medium	Resolved	In `vestedAmount`, the amount vested is incorrectly calculated due to rounding during division, resulting in some amount remaining locked after vesting is completed.
OS-VTVL-ADV-08	Medium	Resolved	`revokeClaim` mishandles the edge case where the user does not claim the amount after vesting is completed.
OS-VTVL-ADV-09	Medium	Resolved	The `vestingRecipients` may take duplicate entries, which results in the same recipient being registered twice.
OS-VTVL-ADV-10	Medium	Resolved	In `withdraw`, the withdrawn amount is not accurately reflected in `numTokensReservedForVesting`, resulting in inconsistent data.
OS-VTVL-ADV-11	Low	Resolved	The inner loop counter in `initializeMilestones` and `initializeAllocations` is incremented twice.

OS-VTVL-ADV-00 [crit] | Inconsistencies In Calculation Of Fee Amount

Description

In `VTVLVesting::_transferToken`, the `_realFeeAmount` parameter passed to the USDC token transfer function is not correctly converted into a USDC amount with six decimal points. This issue may result in a loss of funds for users who are paying the fee in certain instances.

VTVLVesting.sol SOLIDITY

```solidity
function _transferToken(uint256 _amount, uint256 _scheduleIndex) private {
    if (feePercent > 0) {
        uint256 _feeAmount = calculateFee(_amount);
        uint256 _realFeeAmount = (_feeAmount * conversionThreshold) / 100;

        if (pool != address(0)) {

            [...]

        } else {
            tokenAddress.safeTransfer(_msgSender(), _amount);
            IERC20Extented(USDC_ADDRESS).safeTransferFrom(
                msg.sender,
                feeReceiver,
                _realFeeAmount
            );

            emit FeeReceived(
                feeReceiver,
                _realFeeAmount,
                _scheduleIndex,
                address(USDC_ADDRESS)
            );
        }

    [...]
}
```

Moreover, the formula for calculating `_realFeeAmount` seems inconsistent with the `conversionThreshold` parameter, which is being utilized instead of an actual token price. This is especially true in the `else` case of `pool != address(0)` resulting in improper fee calculations.

Proof of Concept

- Calculating `_realFeeAmount` taking `_feeAmount = 10^18`, where `_realFeeAmount = (_feeAmount * conversionThreshold) / 100`.
- Thus, `_realFeeAmount = (10^18 * 30 / 100)` which is `3 * 10^17` or 300 billion USDC.

- Since this value is directly passed to the USDC transfer function, it will attempt to deduct 300 billion from the user's USDC account.

Remediation

Ensure the accurate calculation of the USDC amount by having the output value be represented with six decimal places. Additionally, when the token price is unavailable, the fee should be taken from the token itself.

VTVLVesting.sol SOLIDITY

```solidity
function _transferToken(uint256 _amount, uint256 _scheduleIndex) private {
    if (feePercent > 0) {
        uint256 _feeAmount = calculateFee(_amount);
        uint256 decimals = tokenAddress.decimals();
        uint256 _realFeeAmount = (_feeAmount * conversionThreshold * (10 ** 6))
↪   / (10 ** decimals) / 100;

        if (pool != address(0)) {

            [...]

        } else {
                tokenAddress.safeTransfer(_msgSender(), _amount - _feeAmount);
                tokenAddress.safeTransfer(feeReceiver, _feeAmount);
                emit FeeReceived(
                    feeReceiver,
                    _feeAmount,
                    _scheduleIndex,
                    address(tokenAddress)
                );
        }

    [...]
}
```

Patch

Fixed in commit 5ffecbf by converting the `_realFeeAmount` from the respective token's decimals into USDC decimals.

OS-VTVL-ADV-01 [high] | Incorrect Token Addresses Passed For Quote Price

Description

The calculation of amountOut for the given amount of base token is performed incorrectly in getTokenPrice inside UniswapOracle, where the quote token is USDC.

UnisapOracle.sol SOLIDITY
```solidity
function getQuoteAtTick(
    int24 tick,
    uint128 baseAmount,
    address baseToken,
    address quoteToken
) internal pure returns (uint256 quoteAmount) {

    [...]
}
```

As highlighted above, getQuoteAtTick requires two token addresses as arguments: the baseToken address followed by the quoteToken address. However, in getTokenPrice, a call to getQuoteAtTick occurs with baseToken set to the USDC token address, and the intended base token is mistakenly passed as the quoteToken argument, resulting in an unintended swap of the argument order.

UnisapOracle.sol SOLIDITY
```solidity
function getTokenPrice(
    uint128 amount,
    uint32 secondsAgo
) public view returns (uint amountOut) {
    (int24 tick, ) = consult(secondsAgo);
    amountOut = OracleLibrary.getQuoteAtTick(
        tick,
        amount,
        USDC_ADDRESS,
        address(tokenAddress)
    );

    [...]
}
```

This will return an incorrect price for the quoteToken, resulting in further issues whenever getTokenPrice is called.

Remediation

Ensure passing the correct order of arguments to `getQuoteAtTick`, where the `baseToken` address is passed first and then the `quoteToken` address.

UnisapOracle.sol SOLIDITY

```solidity
function getTokenPrice(
    uint128 amount,
    uint32 secondsAgo
) public view returns (uint amountOut) {
    (int24 tick, ) = consult(secondsAgo);
    amountOut = OracleLibrary.getQuoteAtTick(
        tick,
        amount,
        address(tokenAddress),
        USDC_ADDRESS,
    );

    [...]
}
```

Patch

Fixed in cf25628 by ensuring that the arguments are passed in the correct order to `getQuoteAtTick`.

OS-VTVL-ADV-02 [high] | Unlimited Minting Of Tokens

Description

`VariableSupplyERC20Token::mint` limits the amount of `ERC20` tokens that may be minted. This is tracked via the `mintableSupply` value, which determines the maximum amount of tokens mintable.

To validate that the amount minted does not exceed the max supply of tokens, `mint` reduces `mintableSupply` by the amount being minted. This method is faulty as eventually, as the value of `mintableSupply` decreases and reaches zero, `mint` may effectively mint an unlimited amount of `ERC20` tokens.

VariableSupplyERC20Token.sol 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);
}
```

Remediation

Verify that the sum of the current amount being minted and the total amount minted until now does not surpass `mintableSupply`.

VariableSupplyERC20Token.sol 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(totalSupply + amount <= mintableSupply, "INVALID_AMOUNT");
    }
    _mint(account, amount);
}
```

Patch

Fixed in commit d0716cb by implementing a check to ensure that the sum of `totalSupply` and the newly minting amount does not exceed the `mintableSupply`.

OS-VTVL-ADV-03 [high] | Absence Of Check On Initial Tokens Minted

Description

In the constructor of `VariableSupplyERC20Token`, the `maxSupply_` and `initialSupply_` values are passed in as arguments.

The constructor lacks any check on the value of `initialSupply_` to assert that it is not above the maximum supply of tokens mintable, which is determined in `maxSupply_`. Thus, while deploying this contract, it is possible to assign `initialSupply_` such a value that it exceeds that of `maxSupply_`, minting more tokens than the intended supply.

VariableSupplyERC20Token.sol SOLIDITY

```solidity
constructor(string memory name_, string memory symbol_, uint256 initialSupply_,
↪ uint256 maxSupply_) ERC20(name_, symbol_) {

    require(initialSupply_ > 0 || maxSupply_ > 0, "INVALID_AMOUNT");
    mintableSupply = maxSupply_;

    // Note: the check whether the initial supply is less than or equal to
↪   mintableSupply will happen in mint fn.
    if(initialSupply_ > 0) {
        mint(_msgSender(), initialSupply_);
    }
}
```

Remediation

Incorporate a check in the constructor that validates `initialSupply_` is not greater than `maxSupply_`.

VariableSupplyERC20Token.sol SOLIDITY

```solidity
constructor(string memory name_, string memory symbol_, uint256 initialSupply_,
↪ uint256 maxSupply_) ERC20(name_, symbol_) {

    require(initialSupply_ > 0 || maxSupply_ > 0, "INVALID_AMOUNT");
    mintableSupply = maxSupply_;

    require(initialSupply_ <= maxSupply_, "INITIAL_SUPPLY_EXCEEDS_MAX_SUPPLY");

    // Note: the check whether the initial supply is less than or equal to
↪   mintableSupply will happen in mint fn.
    if(initialSupply_ > 0) {
        mint(_msgSender(), initialSupply_);
    }
}
```


Patch

Fixed in commit d0716cb by implementing a check to ensure that the `initialSupply_` does not exceed the `mintableSupply`.

OS-VTVL-ADV-04 [high] | Inaccurate Price Calculation Formula

Description

`getTokenPrice` in `UniswapOracle` provides the price of one base token (`10**decimal`) in USD with two decimal precision. However, there is an error in the formula for the calculation, resulting in inaccurate price values.

The following is the conversion formula used: `((amountOut * 100) / 10 ** (decimal - 6)) / amount`. The formula is incorrect as the quantity `10 ** (decimal - 6)` acts as the denominator when it should have been multiplied to `(amountOut * 100)`. Moreover, due to the presence of `10 ** (decimal - 6)`, a token with a decimal value less than ten will fail to calculate the price as an underflow error will occur.

UnisapOracle.sol SOLIDITY

```solidity
function getTokenPrice(
    uint128 amount,
    uint32 secondsAgo
) public view returns (uint amountOut) {
    [...]

    // calculate the price with 100 times
    uint256 decimal = IERC20Extented(tokenAddress).decimals();
    return ((amountOut * 100) / 10 ** (decimal - 6)) / amount;
}
```

Proof of Concept

- Consider a token with `decimal = 8` and a price of 20 USD per token.
- Let the amount of the base token be `2 * 10**8`, thus, `amountOut = 40 * 10 ** 6`.
- Substituting in the formula, we get: `((40 * 10**6 * 100) / 10 ** (8 - 6)) / (2 * 10 ** 8)` which amounts to `1/5`, resulting in zero.
- This final result is incorrect as it should have been 2000 (since the price of one token is 20 USD).

Remediation

Pass in the price of only one token (i.e. `10**decimal`) as the `baseAmount` to `getQuoteAtTick`, which will then result in `amountOut` being the USD price of one token in six decimals.

To ensure the result converts to a precision of two decimal places, divide this `amountOut` by `10**4`. This will also result in a more optimized approach by reducing the number of calculations performed.

Patch

Fixed in commit a3d984d by correcting and simplifying the token price calculation formula in `getTokenPrice`.

OS-VTVL-ADV-05 [high] | Locked Tokens In Milestone Contract

Description

In `BaseMilestone::setComplete`, the `OnlyDeposited` modifier allows a `milestone` to be marked as complete if and only if it is deposited fully.

The problem arises from the modifier's method of checking the balance, which checks if it is greater than or equal to `allocation * recipients.length`. As recipients withdraw tokens, the contract's balance reduces. However, `allocation * recipients.length` remains constant. This may result in a situation where the remaining balance intended for allocation becomes locked until more funds are deposited to raise the balance above the limit. As a result, the additional deposited amount for raising the amount will be locked in the contract.

BaseMilestone.sol SOLIDITY

```solidity
modifier onlyDeposited() {
    uint256 balance = tokenAddress.balanceOf(address(this));
    require(balance >= allocation * recipients.length, "NOT_DEPOSITED");
    _;
}
```

Remediation

Remove the constant value, and `numTokensReservedForVesting` should be utilized instead, which tracks all the tokens reserved for vesting in the contract. Also, as milestones complete, `numTokensReservedForVesting` increases with added tokens allocated for vesting thus the following check should occur in `SetComplete`:

BaseMilestone.sol SOLIDITY

```solidity
function setComplete(
    address _recipient,
    uint256 _milestoneIndex
) public onlyOwner {
    Milestone storage milestone = milestones[_recipient][_milestoneIndex];
    require(balanceOf(address(this)) - numTokensReservedForVesting >=
↪   milestone.allocation, "NOT_DEPOSITED")

    require(milestone.startTime == 0, "ALREADY_COMPLETED");

    milestone.startTime = block.timestamp;
    numTokensReservedForVesting += milestone.allocation;
}
```

Patch

Fixed in commit cdfa29d by introducing the `totalWithdrawnAmount` variable to track the withdrawn tokens, which is then subtracted from `numTokensReservedForVesting` (locked tokens).

OS-VTVL-ADV-06 [high] | Inaccurate Withdrawable Token Amount

Description

In `BaseMilestone::withdrawAdmin`, the `availableAmount` is calculated as `allocation * recipients.length - numTokensReservedForVesting`. This may result in locking some tokens that are not reserved for vesting.

BaseMilestone.sol SOLIDITY

```solidity
function withdrawAdmin() public onlyOwner {
    uint256 availableAmount = allocation * recipients.length -
        numTokensReservedForVesting;

    tokenAddress.safeTransfer(msg.sender, availableAmount);

    emit AdminWithdrawn(_msgSender(), availableAmount);
}
```

The issue arises as the contract does not accurately track the actual tokens deposited by each recipient. It only keeps track of the total tokens reserved for vesting (`numTokensReservedForVesting`) across all milestones. Due to this, there may be instances where the contract mistakenly treats some tokens as reserved for vesting even though they have not been assigned to any specific recipient. As a result, these unassigned tokens are effectively locked and may not be withdrawn by the contract owner.

Remediation

Calculate `AvailableAmount` considering both the current contract balance and the amount of tokens reserved for vesting. By doing so, the accurate amount withdrawable will be determined, preventing any unnecessary locking of funds beyond what is already reserved for vesting.

BaseMilestone.sol SOLIDITY

```solidity
function withdrawAdmin() public onlyOwner {
    uint256 availableAmount = balanceOf(address(this)) -
 ↪  numTokensReservedForVesting;

    tokenAddress.safeTransfer(msg.sender, availableAmount);

    emit AdminWithdrawn(_msgSender(), availableAmount);
}
```

Patch

Fixed in commits dbf1f5a and cdfa29d by considering both the contract token balance and the withdrawn token along with the locked tokens for calculating the remaining amount withdrawable.

OS-VTVL-ADV-07 [med] | Flawed Computation Of Current Vested Amount

Description

`vestedAmount` in `VestingMilestone` returns the current amount of tokens that have been vested for a particular user's `milestone`; this is calculated by: `amountPerInterval * intervals`.

Calculation of `amountPerInterval` occurs by dividing the allocated amount by the total number of intervals required to complete vesting. Since this division is rounded down, it may restrict a small amount of tokens from being transferred to the user on claiming after completing the milestone.

VestingMilestone.sol SOLIDITY

```solidity
function vestedAmount(
    address _recipient,
    uint256 _milestoneIndex,
    uint256 _referenceTs
) public view hasMilestone(_recipient, _milestoneIndex) returns (uint256) {
    Milestone memory milestone = milestones[_recipient][_milestoneIndex];
    [...]
    if (_referenceTs > milestone.startTime) {
        uint256 currentVestingDurationSecs = _referenceTs -
            milestone.startTime; // How long since the start

        uint256 intervals = currentVestingDurationSecs /
            milestone.releaseIntervalSecs;
        uint256 amountPerInterval = (milestone.releaseIntervalSecs *
            milestone.allocation) / milestone.period;

        return amountPerInterval * intervals;
    }

    return 0;
}
```

Proof of Concept

- Consider a `milestone` with:
  - `allocation = 10`.
  - `period = 10`.
  - `releaseIntervalSecs = 3`.
- Thus, `amountPerInterval = allocation / total no of intervals`, which comes out to be 10/3, i.e. three.
- After the `milestone` has vested the amount that may be withdrawn is: `amountPerInterval * total intervals`, i.e 3*3, which is nine.

- Recall that our original `allocation` was ten. However, we may only withdraw nine after vesting is complete, thereby locking one token.

Remediation

Modify `vestedAmount` such that it returns `milestone.allocation` when `_referenceTs` is greater than `milestone.startTime + milestone.period` (when `milestone` has completely vested.)

VestingMilestone.sol SOLIDITY

```solidity
    function vestedAmount(
        address _recipient,
        uint256 _milestoneIndex,
        uint256 _referenceTs
    ) public view hasMilestone(_recipient, _milestoneIndex) returns (uint256) {
        Milestone memory milestone = milestones[_recipient][_milestoneIndex];
        [...]

        // Check if this time is over vesting end time
        if (_referenceTs > milestone.startTime + milestone.period) {
            return milestone.allocation;
        }
        [...]

        return 0;
    }
```

Patch

Fixed in commit 8e125ef by modifying `vestedAmount` to return `milestone.allocation` in the case of milestone completion.

OS-VTVL-ADV-08 [med] | Incorrect Check For Unvested Amount

Description

In `VTVLVesting::revokeClaim`, `amountWithdrawn` for the claim is compared to `finalVestAmt` to prevent the revocation of a claim that has been fully consumed.

This method will fail when a user has not claimed the amount after the vesting was completed, as even in this case, `_claim.amountWithdrawn < finalVestAmt` will result in `true`.

VTVLVesting.sol SOLIDITY

```solidity
function revokeClaim(
    address _recipient,
    uint256 _scheduleIndex
) external onlyOwner hasActiveClaim(_recipient, _scheduleIndex) {
    // Fetch the claim
    Claim storage _claim = claims[_recipient][_scheduleIndex];

    // Calculate what the claim should finally vest to
    uint256 finalVestAmt = finalVestedAmount(_recipient, _scheduleIndex);

    // No point in revoking something that has been fully consumed
    // so require that there be unconsumed amount
    require(_claim.amountWithdrawn < finalVestAmt, "NO_UNVESTED_AMOUNT");

    [...]
}
```

Remediation

Replace the current verification of the consumed amount utilizing `_claim.amountWithdrawn` with `vestedSoFarAmt` in the required statement. This modification will correctly handle situations where the user does not withdraw the amount, but the vesting period has elapsed, as `vestedSoFarAmt` will always be greater than or equal to `finalVestAmt`.

VTVLVesting.sol SOLIDITY

```solidity
function revokeClaim(
    address _recipient,
    uint256 _scheduleIndex
) external onlyOwner hasActiveClaim(_recipient, _scheduleIndex) {
    // Fetch the claim
    Claim storage _claim = claims[_recipient][_scheduleIndex];

    // Calculate what the claim should finally vest to
    uint256 finalVestAmt = finalVestedAmount(_recipient, _scheduleIndex);
```

```
        uint256 vestedSoFarAmt = vestedAmount(
            _recipient,
            _scheduleIndex,
            uint40(block.timestamp)
        );

        // No point in revoking something that has been fully consumed
        // so require that there be unconsumed amount
        require(vestedSoFarAmt < finalVestAmt, "NO_UNVESTED_AMOUNT");

        [...]
    }
```

Patch

Fixed in commits cfb17bc and 56f3e40 by allowing the withdraw function to be utilized on revoked claims, enabling the claimants to retrieve the remaining vested amount that was not withdrawn before the claim was revoked.

OS-VTVL-ADV-09 [med] | Duplicate Entries In Vesting Recipients Array

Description

`VTVLVesting::_createClaimUnchecked` updates the `vestingRecipients` array with the recipients' address of a particular vesting to track the recipients of the vesting in the future. This `vestingRecipients` array may currently hold duplicate recipients' addresses, resulting in inconsistencies as each unique address should have only one distinct claim against it.

VTVLVesting.sol SOLIDITY

```solidity
function _createClaimUnchecked(ClaimInput memory claimInput) private {
    [...]

    vestingRecipients.push(claimInput.recipient); // add the vesting recipient
↪   to the list

    [...]
}
```

Remediation

Incorporate a check at the very beginning of this function that ensures that the passed-in recipient address does not already exist in the `claims` mapping.

VTVLVesting.sol SOLIDITY

```solidity
function _createClaimUnchecked(ClaimInput memory claimInput) private {
    require(claims[claimInput.recipient] == 0, "RECIPIENT_ALREADY_EXISTS");

    [...]

    vestingRecipients.push(claimInput.recipient); // add the vesting recipient
↪   to the list

    [...]
}
```

Patch

Fixed in commit 5880740 by removing the `vestingRecipients` storage variable.

OS-VTVL-ADV-10 [med] | Insufficient Logging Of Withdrawn Amount

Description

In VestingMilestone and SimpleMilestone, withdraw enables the recipient to withdraw all remaining claimable amounts from their active milestone.

The issue arises due to how the function reflects this change in token balance. The withdrawn amount is updated in milestone.withdrawnAmount by setting its value to that of the total allowance in VestingMilestone, and in SimpleMilestone, milestone.isWithdrawn is set to true.

However, in both modules, the change is not reflected in numTokensReservedForVesting, representing the current total amount of tokens reserved for vesting and not yet withdrawn. Thus, the amount of numTokensReservedForVesting remains static even after the withdrawal.

Milestone Withdrawal SOLIDITY

```solidity
// in SimpleMilestone.sol
function withdraw(
    uint256 _milestoneIndex
)   public
    hasMilestone(_msgSender(), _milestoneIndex)
    onlyCompleted(_msgSender(), _milestoneIndex)
{
    [...]

    milestone.isWithdrawn = true;
    tokenAddress.safeTransfer(_msgSender(), milestone.allocation);

}

// in VestingMilestone.sol
function withdraw(
    uint256 _milestoneIndex
)
    external
    hasMilestone(_msgSender(), _milestoneIndex)
    onlyCompleted(_msgSender(), _milestoneIndex)
{
    [...]

    milestone.withdrawnAmount = allowance;

    tokenAddress.safeTransfer(_msgSender(), amountRemaining);

}
```

This results in inconsistent data, where the withdrawn amount is not properly accounted for, which may result in potential double counting or improper allocation of tokens.

Remediation

Ensure in both modules, the amount transferred to the user in `withdraw` is reduced from the tokens reserved for vesting (i.e. `numTokensReservedForVesting`), thus properly reflecting the withdrawn amount in contract storage.

Milestone Withdrawal SOLIDITY

```
// in SimpleMilestone.sol
function withdraw(
    uint256 _milestoneIndex
)   public
    hasMilestone(_msgSender(), _milestoneIndex)
    onlyCompleted(_msgSender(), _milestoneIndex)
{
    [...]

    milestone.isWithdrawn = true;
    numTokensReservedForVesting -= milestone.allocation;

    tokenAddress.safeTransfer(_msgSender(), milestone.allocation);

}

// in VestingMilestone.sol
function withdraw(
    uint256 _milestoneIndex
)
    external
    hasMilestone(_msgSender(), _milestoneIndex)
    onlyCompleted(_msgSender(), _milestoneIndex)
{
    [...]

    milestone.withdrawnAmount = allowance;
    numTokensReservedForVesting -= amountRemaining;

    tokenAddress.safeTransfer(_msgSender(), amountRemaining);

}
```

Patch

Fixed in commit cdfa29d by introducing the `totalWithdrawnAmount` variable to track the withdrawn tokens.

OS-VTVL-ADV-11 [low] | Issue In Loop Increments

Description

In `BaseMilestone`, both `initializeMilestones` and `initializeAllocations` have two loops. The outer loop iterates over all the `milestones`, while the inner loop iterates over the recipients' addresses. However, the loop counter of the inner loop increments twice: once in the loop header and again within the loop body, inside the `unchecked` block.

This double incrementation skips every other recipient's address, resulting in some addresses not being assigned any `milestone` and allocation, resulting in inconsistencies.

BaseMilestone.sol SOLIDITY

```solidity
    function initializeMilestones(
        InputMilestone[] memory _milestones
    ) internal {
        uint256 length = _milestones.length;

        for (uint256 i = 0; i < length; ) {
            [...]

            for (uint256 j = 0; j < recipientLenth; j++) {
                milestones[recipients[j]][i] = milestone;
                unchecked {
                    ++j;
                }
            }
            [...]

        }
    }

    function initializeAllocations(
        uint256[] memory _allocationPercents
    ) internal {
        for (uint256 i = 0; i < length; ) {
            [...]

            for (uint256 j = 0; j < recipientLenth; j++) {
                unchecked {
                    milestones[recipients[j]][i].allocation = amount;
                    ++j;
                }
            }
            [...]

        }
    }
```

Remediation

To ensure accurate loop increments, eliminate the counter increment operation from the loop header of the inner loop in both functions.

BaseMilestone.sol SOLIDITY

```solidity
    function initializeMilestones(
        InputMilestone[] memory _milestones
    ) internal {
        uint256 length = _milestones.length;

        for (uint256 i = 0; i < length; ) {
            [...]

            for (uint256 j = 0; j < recipientLenth; ) {
                milestones[recipients[j]][i] = milestone;
                unchecked {
                    ++j;
                }
            }
            [...]

        }
    }

    function initializeAllocations(
        uint256[] memory _allocationPercents
    ) internal {
        for (uint256 i = 0; i < length; ) {
            [...]

            for (uint256 j = 0; j < recipientLenth; ) {
                unchecked {
                    milestones[recipients[j]][i].allocation = amount;
                    ++j;
                }
            }
            [...]

        }
    }
```

Patch

Fixed in commit a8bf6da by removing the counter increment operation from the loop header of the inner loop in both functions.

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 lead to security issues in the future.

ID	Description
OS-VTVL-SUG-00	In `calculateFee`, the fee value calculated is inherently rounded down instead of rounding up.
OS-VTVL-SUG-01	In `AccessProtected`'s constructor, `tx.orogin` is assigned as an admin, which is unsafe and may have adverse effects.
OS-VTVL-SUG-02	`consult` in `UniswapOracle` is redundant.
OS-VTVL-SUG-03	Utilize re-entrancy guards to prevent re-entrancy attacks.
OS-VTVL-SUG-04	Modify the calculation of reference timestamps to increase accuracy.
OS-VTVL-SUG-05	Create a check to assert the amount being minted is non-zero in `VariableSupplyERC20Token`.
OS-VTVL-SUG-06	Setting limits for the `feePercent` parameter will ensure it does not become extremely high or low.

OS-VTVL-SUG-00 | Round Up Fee Value

Description

In `VTVLVesting::calculateFee`, the fee value is automatically rounded down due to the default solidity behavior for division.

VTVLVesting.sol SOLIDITY

```solidity
function calculateFee(uint256 _amount) private view returns (uint256) {
    return (_amount * feePercent) / 10000;
}
```

Remediation

Round up the division, as it ensures that the fee is always slightly overestimated, avoiding the presence of any fractional parts in fee calculations.

OS-VTVL-SUG-01 | Check For Admin Address

Description

In the constructor of `AccessProtected`, `tx.origin` is set as the admin. If the intention of utilizing `tx.origin` is to restrict smart contracts from calling and only allow externally owned accounts to call, it is advisable to use `msg.sender`.

AccessProtected.sol SOLIDITY

```solidity
    constructor() {
        _admins[tx.origin] = true;
        emit AdminAccessSet(tx.origin, true);
    }
```

This is because `tx.origin` is highly unsafe as it leaves the contract vulnerable to phishing-like attacks, as `tx.origin` is always the address that first initialized the call. If there are multiple calls after this, it will not be reflected in `tx.origin` with the result that the address that initiated the final call may be a smart contract.

Remediation

Utilize `msg.sender` and `tx.origin` to ensure that the constructor is only able to be called by an externally owned account by checking if `msg.sender` is equal to `tx.origin`. This ensures that the address which initiated the transaction is also the address calling it.

AccessProtected.sol SOLIDITY

```solidity
    constructor() {
        require(msg.sender == tx.origin, "NOT_AN_EOA");
        _admins[tx.origin] = true;
        emit AdminAccessSet(tx.origin, true);
    }
```

OS-VTVL-SUG-02 | Redundant Functionality

Description

In `UniswapOracle`, `getTokenPrice` utilizes `consult` to receive the time-weighted means of tick and liquidity for a given Uniswap V3 pool. This is a redundant functionality as it is already present in `OracleLibrary::consult`, imported by `UniswapOracle`.

Remediation

To optimize gas consumption and make the code more consistent and maintainable, remove `consult` in `UniswapOracle`. Instead, utilize `OracleLibrary::consult`.

OS-VTVL-SUG-03 | Mitigating Re-entrancy

Description

To prevent potential re-entrancy attacks, utilize a re-entrancy guard in the following functions that make external contract calls and accept inputs from users:

- `VTVLMilestoneFactory::createVestingMilestone`
- `VTVLMilestoneFactory::createSimpleMilestones`
- `VTVLVesting::createClaim`
- `VTVLVesting::createClaimsBatch`
- `BaseMilestone::withdrawAdmin`
- `BaseMilestone::deposit`
- `SimpleMilestone::withdraw`
- `VestingMilestone::withdraw`

Ensure the above functions utilize the `ReentrancyGuard` modifier.

OS-VTVL-SUG-04 | Accurate Calculation Of Vested Amount

Description

In VTVLVesting, vestedAmount returns the amount vested for a given _recipient at a reference timestamp. It currently selects the reference time stamp as either the user-given timestamp if the claim is active or the deactivationTimestamp if the claim is inactive.

VTVLVesting.sol SOLIDITY

```solidity
    function vestedAmount(
        address _recipient,
        uint256 _scheduleIndex,
        uint40 _referenceTs
    ) public view returns (uint256) {
        Claim memory _claim = claims[_recipient][_scheduleIndex];
        uint40 vestEndTimestamp = _claim.isActive
            ? _referenceTs
            : _claim.deactivationTimestamp;
        return _baseVestedAmount(_claim, vestEndTimestamp);
    }
```

Thus, an inactive claim would not return the correct vested amount at the specified time as deactivationTimestamp is taken instead.

Remediation

Utilize the minimum value between _referenceTs and deactivationTimestamp as the timestamp in the calculation. This approach guarantees the function returns the correct vested amount, even if the claim is inactive.

OS-VTVL-SUG-05 | Check For Zero Amount Minting

Description

`mint` in `VariableSupplyERC20Token` allows minting of zero amounts. This may be misleading and result in invalid minting of tokens and is generally unfavorable.

Remediation

Ensure that `mint` checks the amount passed in for minting, asserting that it is not zero.

VariableSupplyERC20Token.sol SOLIDITY

```
function mint(address account, uint256 amount) public onlyAdmin {
    require(account != address(0), "INVALID_ADDRESS");
    require(amount != 0, "INVALID_AMOUNT");
    // 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);
}
```

OS-VTVL-SUG-06 | Set Bounds For Fee Percent

Description

In `VTVLVestingFactory::createVestingContract`, the `feePercent` value may be set to any value as it is not restricted. This may result in the `feePercent` becoming very high or low. Both these cases are unfavorable and will have adverse outcomes, affecting the user experience.

Remediation

Incorporate checks to ensure the `feePercent` parameter is within suitable upper and lower bounds, restricting the possibility of setting the `feePercent` parameter to extreme values.

A | 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

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 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.