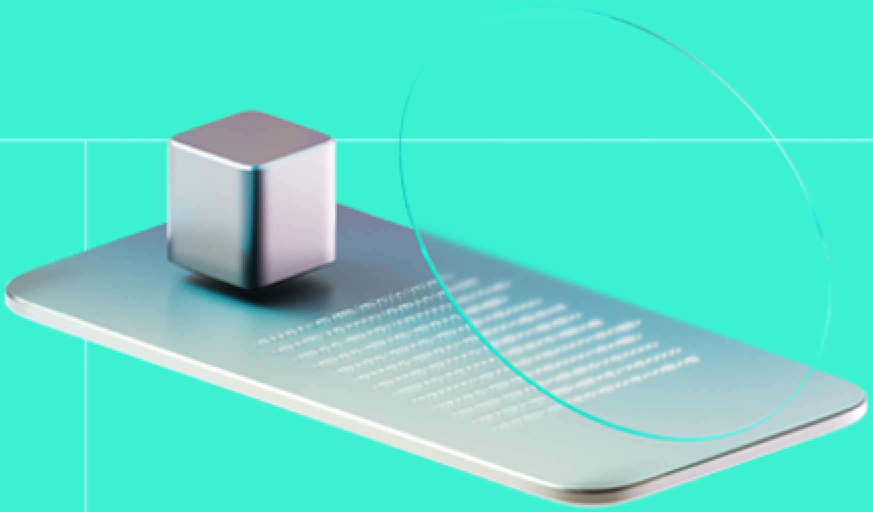
Smart Contract Code Review And Security Analysis Report



Customer: Jellyverse

Date: 17/05/2024

We express our gratitude to the Jellyverse team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Jellyverse represents an expansive ecosystem within the decentralized finance (DeFi) landscape, managed by its community via a decentralized autonomous organization (DAO). It is dedicated to advancing the latest version of DeFi, often referred to as DeFi 3.0, with a vision to establish a sustainable, yield-focused framework that merges protocols with tangible real-world assets. The platform offers multiple utilities crafted to address diverse financial requirements, all regulated by its native cryptocurrency, JLY.

The governance model of Jellyverse, called Governor, is designed to enable users to steer protocols within its ecosystem effectively. Additionally, it allows new initiatives the opportunity to become part of Jellyverse. The system facilitates a fair allocation of protocol earnings to stakers, positioning it as an attractive option for projects seeking to prosper.

Platform: EVM

Language: Solidity

Tags: Governance, Vesting

Timeline: 18/04/2024 - 26/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/MVPWorkshop/jelly-verse-contracts/tree/main
Commit	3331a1e

Audit Summary

10/10	9/10	95,52%	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.6/10

The system users should acknowledge all the risks summed up in the risks section of the report.

10	6	4	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	2
High	0
Medium	4
Low	4

Findings	Status
F-2024-1509 - Voting power restored without cost after cliff period	Accepted
F-2024-1510 - Missing fee validation in setFee function	Accepted
F-2024-1512 - Proposals are vulnerable to metamorphic attacks	Accepted
F-2024-1559 - Lack of upper bound Beneficiaries may lead to deny of service	Accepted
F-2024-1492 - Multiple unstake at once possible due to _releasableAmount incorrect calculations	Fixed
F-2024-1497 - Any voting can be abused due to Chest ids duplicates passing possibility	Fixed
F-2024-1511 - Overestimated voting power in first freezing week	Fixed
F-2024-1547 - Voting power restored with minimal cliff period extension	Fixed
F-2024-1560 - claimWeeks does not burn tokens in StakingRewardDistribution	Fixed
F-2024-1563 - dailySnapshot can revert due to integer underflow	Fixed

Document

Name	Smart Contract Code Review and Security Analysis Report for Jellyverse
Audited By	Jakub Heba, Ataberk Yavuzer
Approved By	Ataberk Yavuzer
Website	https://jellyverse.org
Changelog	30/04/2024 - Preliminary Report
	17/05/2024 - Remediation Check

Table of Contents

System Overview

Jellyverse is a DeFi platform, managed by the governance, with multiple utilities, such as staking, vesting, native tokens to jelly tokens swapping, with its own JLY ERC20 token implementation, working as governance and utility token.

Chest.sol - staking contract representing user voting power by means of chest. Voting power decreases over time and upon unstaking.
DailySnapshot.sol - a contract to store and retrieve daily snapshots blocknumbers per epoch.
Governor.sol - it is a decentralised voting for proposals solution. It is assigned to signle Chest instance, to determine stakers voting power. It is modular, can be extended by JellyGovernor and other submodules: GovernorCountingSimple, GovernorSettings, GovernorTimelockControl.
GovernorVotes - calculates user's votes based on chests staked.
InvestorDistribution - a contract that stakes chests for predefined investors.
JellyGovernor - extension of Governor, which specifies voting function and quorum.
JellyTimelock - an instance of TimelockController. Connected with governance to execute proposals timely.
JellyToken - an ERC20 token representing the main token of Jelly ecosystem.
JellyTokenDeployer - a factory contract that allows to deploy JellyToken instace via Create2.
LiquidityRewardDistribution - a contract for distributing liquifity mining rewards.
Minter - a contract that mints Jelly tokens timely for beneficiaries.
OfficialPoolsRegister - a registry contract to store and retrieve official DEX pools.
PoolParty - a contract for swapping native tokens for jelly tokens. The received payment in USD stable coin is later used to provide liquidity for DEX pool.
RewardVesting - a contract designed for claiming vested liquidity. It is used by the LiquidityRewardDistribution.
StakingRewardDistribution - a contract for distributing staking rewards.
TeamDistribution - a contract that stakes chests for predefined team members.

Privileged roles

- MINTER_ROLE - in JellyToken contract, this actor could perform initial preminting of tokens to specified addresses, as well as is able to mint JLY tokens through the mint() function
- Owner of Chest contract is able to set arbitrary fee value, which could have negative consequences for users using the platform. However, this address will be manager by governance, so risk of it is much lowered.
- Owner of LiquidityRewardDistribution, as well as StakingRewardDistribution, could change vesting contract address which could be dangerous if replaced with malicious one.
- Owner of PoolParty can set up _ustToJellyRatio arbitrary, which may lead to a situation where this ratio is set very high value, what could be problematic for users using the buyWithUsd() function

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are covered.
- Technical description are provided for most critical parts..

Code quality

The total Code Quality score is **9** out of **10**.

- Several template code patterns were found.
- Code lacks explanations for more advanced operations and math calculations

Test coverage

Code coverage of the project is **76.5%** (branch coverage), with a mutation score of **95.52%**.

- Deployment and basic user interactions are covered with tests.
- Edge case scenarios and negative cases coverage are missed somethimes

Security score

Upon auditing, the code was found to contain **2** critical, **0** high, **4** medium, and **4** low severity issues.

All identified issues are detailed in the "Findings" section of this report.

After performing the remediation check, it was found that all findings have been resolved or their risk has been accepted, what reflects into 0 pending issues. Therefore, the security score is **10** out of **10**..

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.6**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- If much more time passes between `block.timestamp` and `beginningOfTheNewDayTimestamp` than expected - the calculations of the time of subsequent snapshots will not be accurate, because they will not refer to the current time, but to an old, distant value.
- In current implementation, `GovernorCountingSimple` in `_quorumReached` function is not calculating votes against, what is problematic, as far as for calculations of Quorum all votes should be taken into account.
- In `PoolParty` protocol assumes that the value of the JPY token will be less than one dollar, which if it exceeds it will make the calculations incorrect

Findings

Finding Details

F-2024-1492 - Multiple unstake at once possible due to _releasableAmount incorrect calculations - Critical

Description:

During the `unstake` operation, the `_releasableAmount` function is called. It calculates the amount that has already vested but hasn't been released yet.

For amount calculation, it checks where we are in the process. Three scenarios are possible:

- Current `timestamp` is less than `vestingPosition.cliffTimestamp`
- Current `timestamp` is greater than or equal to `vestingPosition.cliffTimestamp + vestingPosition.vestingDuration`
- Current `timestamp` is greater than `vestingPosition.cliffTimestamp` but less than `vestingPosition.cliffTimestamp + vestingPosition.vestingDuration`

In the last scenario, the amount is calculated at `utils/Vesting.sol#69` according to the following formula:

```
amount =
((vestingPosition.totalVestedAmount -
vestingPosition.releasedAmount) *
(block.timestamp - vestingPosition.cliffTimestamp)) /
vestingPosition.vestingDuration;
```

This calculation seems to be incorrect because `vestingPosition.releasedAmount` should be in a different place. The corrected formula:

```
amount =
((vestingPosition.totalVestedAmount) *
(block.timestamp - vestingPosition.cliffTimestamp)) /
vestingPosition.vestingDuration -
vestingPosition.releasedAmount;
```

As a consequence, a user calling the `unstake` function, for example twice in a row, receives an inappropriate amount of tokens, taking into account a given point in the vesting time.

Assets:

- utils/Vesting.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Fixed

Classification

Impact:	5/5
Likelihood:	5/5
Exploitability:	Independent
Complexity:	Medium
Severity:	Critical

Recommendations

Remediation: We suggest changing the `amount` value calculation for the scenario when current `timestamp` is greater than `vestingPosition.cliffTimestamp` but less than `vestingPosition.cliffTimestamp + vestingPosition.vestingDuration`.

Resolution: The Finding was fixed in commit `d5df9d8`, by adjusting the mathematical operation to the correct order.

Evidences

Proof of Concept

Reproduce:

```
function test_unstake_misscalculation() external openSpecialPosition {
uint256 positionIndex = 0;
Chest.VestingPosition memory vestingPositionBefore = chest
.getVestingPosition(positionIndex);

uint256 accountJellyBalanceBefore = jellyToken.balanceOf(testAddress);
uint256 chestJellyBalanceBefore = jellyToken.balanceOf(address(chest));

vm.warp(vestingPositionBefore.cliffTimestamp + 1);

uint256 unstakeAmount = chest.releasableAmount(0);

// First releasableAmount calculation
uint256 releasableAmount = chest.releasableAmount(0);
assertEq(releasableAmount, 1 ether);

// First unstake
vm.prank(testAddress);
chest.unstake(positionIndex, unstakeAmount);

// Second releasableAmount calculation
releasableAmount = chest.releasableAmount(0);
assertEq(releasableAmount, 999000000000000000);

// Second unstake
vm.prank(testAddress);
chest.unstake(positionIndex, releasableAmount);

// If no revert, then second unstake succeed
Chest.VestingPosition memory vestingPositionAfter = chest
.getVestingPosition(positionIndex);

// Here is the proof, that releasedAmount is not equal to unstakeAmount
, because extra 999000000000000000 was unstaked
assertEq(vestingPositionAfter.releasedAmount, unstakeAmount);

}
```

Results:	Test is failing, because user unstaked more, than he should be able to.

F-2024-1497 - Any voting can be abused due to Chest ids duplicates passing possibility - Critical

Description: Chest contract implements `getVotingPower` function, responsible for calculating the voting power of all account's chests. Due to the fact, that `getVotingPower` takes `uint256[] calldata tokenIds` as parameter, and there is no extra logic used for checking, if these values are not duplicated in the table, there is a possibility of multiplication of voting power, by specifying the same `tokenIds` multiple times., consequently, abusing any proposal voting.

Chest.sol#390:

```
function getVotingPower(
address account,
uint256[] calldata tokenIds
) external view returns (uint256) {
uint256 power;
for (uint256 i = 0; i < tokenIds.length; ) {
if (ownerOf(tokenIds[i]) != account) {
revert Chest__NotAuthorizedForToken();
}
power += getChestPower(tokenIds[i]);

unchecked {
++i;
}
}
return power;
}
```

Assets:

- Chest.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Critical

Recommendations

Remediation: We suggest implementing validation of the `tokenIds` parameter in terms of passing duplicates in its content. Only unique values should be filtered, which will then be subject to ownership verification.

Resolution: The Finding was fixed in commit `d5df9d8`, by implementing another `for` loop verifying whether a given object is not duplicated.

Evidences

Proof of Concept

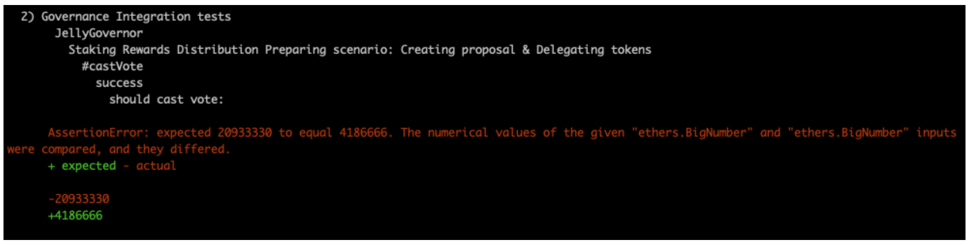
Reproduce:

```
import { assert, expect } from 'chai';
import { time } from '@nomicfoundation/hardhat-network-helpers';
import { utils, BigNumber, constants } from 'ethers';
import { ProposalState, VoteType } from '../../../shared/types';
import { StandardMerkleTree } from "@openzeppelin/merkle-tree";

export function shouldCastVotes(): void {
context(
'Staking Rewards Distribution Preparing scenario: Creating proposal & D
elegating tokens',
async function () {
const proposalDescription = 'Test Proposal #1: Create epoch';
let aliceValues: any[];
let merkleTree;
let merkleRoot;
const ipfsUri = "";
let createEpochFunctionCalldata: string;
let proposalId: BigNumber;
let proposalParams: string;
const chestIDs: number[] = [0, 1, 2];
beforeEach(async function () {
aliceValues = [this.signers.alice.address, "1000"];
merkleTree = StandardMerkleTree.of([aliceValues], ['address', 'uint256'
]);
merkleRoot = merkleTree.root;
createEpochFunctionCalldata = this.stakingRewardDistribution.interface.
encodeFunctionData(
'createEpoch',
[merkleRoot, ipfsUri]
);
await this.jellyGovernor
.connect(this.signers.alice)
.proposeCustom(
[this.stakingRewardDistribution.address],
[0],
[createEpochFunctionCalldata],
proposalDescription,
"3600", /* 1hour in seconds */
"86400" /* 1 day in seconds */
);

const abiEncodedParams = utils.defaultAbiCoder.encode(
['address[]', 'uint256[]', 'bytes[]', 'bytes32'],
[
[this.stakingRe
```

See more

Results: Test will fail, if Alice voting power is much bigger than it should, due to duplicate entries passed in the `proposalParams`.

Files:



```
  2) Governance Integration tests
       JellyGovernor
         Staking Rewards Distribution Preparing scenario: Creating proposal & Delegating tokens
           #castVote
             success
               should cast vote:

      AssertionError: expected 20933330 to equal 4186666. The numerical values of the given "ethers.BigNumber" and "ethers.BigNumber" inputs
were compared, and they differed.
      + expected - actual

      -20933330
      +4186666
```

F-2024-1511 - Overestimated voting power in first freezing week - Medium

Description:

In the current implementation of the contract, calculations based on the `regularFreezingTime` and `_calculateBooster` parameters slightly overestimate `votingPower` in the first week of vesting.

Due to the fact that `regularFreezingTime` causes the further we are from the vesting cliff, the more `votingPower` we have, and at the same time `_calculateBooster` causes the more weeks that have passed, the greater the bonus to `votingPower`, the calculations in the first week are a bit mutually exclusive.

This results in a situation where `votingPower` is slightly higher for the first week than after a full week, which may be unfair in the context of voting.

Chest.sol#559:

```
function _calculatePower(
uint256 timestamp,
VestingPosition memory vestingPosition
) internal pure returns (uint256) {
uint256 power;

uint256 vestingDuration = vestingPosition.vestingDuration;
uint256 cliffTimestamp = vestingPosition.cliffTimestamp;
uint256 unfreezeTime = cliffTimestamp + vestingDuration;

// chest is open, return 0 power
if (timestamp > unfreezeTime) {
return 0;
}

// calculate regular freezing time in weeks
uint256 regularFreezingTime = (cliffTimestamp > timestamp)
? Math.ceilDiv(cliffTimestamp - timestamp, TIME_FACTOR)
: 0;

// calculate power based on vesting type
if (vestingPosition.vestingDuration == 0) {
// regular chest
uint120 booster = _calculateBooster(
vestingPosition,
uint48(timestamp)
);
power =
(booster *
(vestingPosition.totalVestedAmount -
vestingPosition.releasedAmount) *
regularFreezingTime) /
(MIN_STAKING_AMOUNT * DECIMALS); // @dev scaling because of minimum sta
king amount and booster
} else {
// special chest
uint256 linearFreezingTime;
if (timestamp < cliffTimestamp) {
// before the cliff, linear freezing time remains constant
linearFreezingTime =
Math.ceilDiv(vestingDuration, TIME_FACTOR) /
2;
} else {
// after the cliff, linear freezing time starts to decrease
linearFreezingTime =
Math.ceilDiv(unfreezeTime - timestamp, TIME_FACTOR) /
2;
}

// calculate total freezing time in weeks
uint256 totalFreezingTimeInWeeks = regularFreezingTime +
linearFreezingTime;
// apply nerf parameter
uint8 nerfParameter = vestingPosition.nerfParameter;
power =
((vestingPosition.totalVestedAmount -
vestingPosition.releasedAmount) *
totalFreezingTimeInWeeks *
nerfParameter) /
(NERF_NORMALIZATION_FACTOR * MIN_STAKING_AMOUNT); // @dev scaling becau
se of minimum staking amount
}
return power;
}
```

Assets:

- Chest.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Fixed

Classification

Impact:	1/5
Likelihood:	2/5
Exploitability:	Independent
Complexity:	Medium
Severity:	Medium

Recommendations

Remediation:

The logic responsible for calculating the correct voting power should be adjusted so as not to favor new vesting users over those who have already spent at least the first week.

If this is not possible to achieve by adjusting the existing parameters, we suggest implementing a factor that reduces voting power during the first week so that `votingPower` in the first and second weeks are as close as possible to each other.

Resolution:

The Finding was fixed in commit `0374791` by removal of the booster effect for the first week of vesting.

Evidences

Proof of Concept

Reproduce:

```
function test_overestimation_in_first_week() external {
vm.warp(1713695934);

uint256 amount = MIN_STAKING_AMOUNT * 100;
uint32 freezingPeriod = MIN_FREEZING_PERIOD * 10;
vm.startPrank(testAddress);
jellyToken.approve(address(chest), amount + chest.fee());

// We are staking `amount` tokens for `freezingPeriod` of time
chest.stake(amount, testAddress, freezingPeriod);

vm.stopPrank();

uint256 positionIndex = 0;
Chest.VestingPosition memory vestingPosition = chest.getVestingPosition
(
positionIndex
);

uint256 powerHarness = chestHarness.exposed_calculatePower(
block.timestamp,
vestingPosition
);
uint256 powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

assertEq(1000, powerGetter);

// Let's jump to the end of the first week -1 second, so we are still i
n the first week of vesting
vm.warp(
vestingPosition.cliffTimestamp - MIN_FREEZING_PERIOD * 9 - 1
);

powerHarness = chestHarness.exposed_calculatePower(
block.timestamp,
vestingPosition
);
powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

// Voting power is 1006
assertEq(1006, powerGetter);

// Let's jump one second more, so exactly to the beginning of second we
ek
vm.warp(
vestingPosition.cliffTimestamp - MIN_FREEZING_PERIOD * 9
);

powerHarness = chestHarness.exposed_calculatePower(
block.timestamp,
vestingPosition
);
powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

// Voting power is now 905, so much more
assertEq(905, powerGetter);
}
```

Results:

If test passes, then difference between end of the first week, and beginning of the second week, in terms of voting power is more than 10%.

F-2024-1512 - Proposals are vulnerable to metamorphic attacks - Medium

Description:	Proposals are vulnerable to metamorphic attacks where `create2()/selfdestruct()` are used to completely re-write proposal actions right before execution. The main reason for the vulnerability is the fact that `Timelock` does not verify whether the contract code it calls is not different from the code of the same contract at the time the proposal was created.
Assets:	• Governor.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]
Status:	Accepted

Classification

Impact:	4/5
Likelihood:	1/5
Exploitability:	Semi-Dependent
Complexity:	Complex
Severity:	Medium

Recommendations

Remediation:

We suggest saving the hash of the contract code when the proposal is created, and, if it is accepted, comparing it with the contract that will be invoked. If these values are different, execution should be interrupted.

Remediation (Accepted): This finding was acknowledged by the JellyVerse team with the comment:

> In an effort to not add additional constraints, we kept the implementation as is. It is on the voting participants to make sure that the contract targeted in the proposal either does not contain a selfdestruct method, making such an attack impossible, or is owned by the governance contract itself.

F-2024-1547 - Voting power restored with minimal cliff period extension - Medium

Description:

Following the logic of the `Chest` contract, voting power declines over time. However, for regular `Chest`, if we are right before cliff time, there is a bypass possible, to obtain a boost for voting power, which could be used, for example, right before proposal ending for voting.

If `vestingPosition freezingPeriod` is almost finished, so we are very close to the `cliffTimestamp`, voting power is very reduced, comparing to the beginning of vesting process. There are two situations, when users can increase their stake:

- When chest is open, so `block.timestamp ≥ vestingPosition.cliffTimestamp`
- When chest is frozen, so `block.timestamp < vestingPosition.cliffTimestamp`

While for #1 situation there is a validation performed, if `extendFreezingPeriod` is smaller than `MIN_FREEZING_PERIOD`, in the second this check is missing. As a consequence, right before cliff timestamp, users can increase their voting power via `increaseStake` execution with small amount of time in `extendFreezingPeriod`, and any `amount` of tokens.

Chest.sol#251:

```
if (vestingPosition.vestingDuration == 0) {
// regular chest
if (block.timestamp < vestingPosition.cliffTimestamp) {
// chest is frozen
newCliffTimestamp =
vestingPosition.cliffTimestamp +
extendFreezingPeriod;

if (
newCliffTimestamp >
block.timestamp + MAX_FREEZING_PERIOD_REGULAR_CHEST
) {
revert Chest__InvalidFreezingPeriod();
}
vestingPositions[tokenId].cliffTimestamp = newCliffTimestamp;
} else {
// chest is open
if (extendFreezingPeriod < MIN_FREEZING_PERIOD) {
// @dev when chest is open, freezing period must be set to non-zero val
ue and >= MIN_FREEZING_PERIOD
revert Chest__InvalidFreezingPeriod();
}
if (
vestingPosition.totalVestedAmount -
vestingPosition.releasedAmount +
amount <
MIN_STAKING_AMOUNT
) revert Chest__InvalidStakingAmount();
```

Assets:

- Chest.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Fixed

Classification

Impact:	3/5
Likelihood:	2/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation: The logic of the `Chest` contract should be adjusted so that a validation of `extendFreezingPeriod` parameter to be equal or bigger than `MIN_FREEZING_PERIOD` is enforced every time, even before cliffTimestamp is reached.

Resolution: The Finding was fixed in commit `e6edaaf` by validation that `current timestamp + MIN_FREEZING_PERIOD` is not less than `newCliffTimestamp`.

Evidences

Proof of Concept

Reproduce:

```
function test_regular_getChestPower_after_increaseStake_before_cliff()
external {
vm.warp(1713695934);

uint256 amount = MIN_STAKING_AMOUNT * 100;
uint32 freezingPeriod = MIN_FREEZING_PERIOD * 10;

vm.startPrank(testAddress);
jellyToken.approve(address(chest), amount + chest.fee());

// We are staking our amount of tokens
chest.stake(amount, testAddress, freezingPeriod);

vm.stopPrank();

uint256 positionIndex = 0;
Chest.VestingPosition memory vestingPosition = chest.getVestingPosition
(
positionIndex
);

uint256 powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);
// We have full voting power now
assertEq(1000, powerGetter);

// We are teleporting to the moment before cliff
vm.warp(
vestingPosition.cliffTimestamp -1
);

powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

// Voting power is reduced to very low amount, due to the fact that we
are before cliff
assertEq(106, powerGetter);

vm.prank(testAddress);
jellyToken.mint(1_000 * MIN_STAKING_AMOUNT);

vm.startPrank(testAddress);
jellyToken.approve(address(chest), amount*10);

// We are increasing our stake, adding value 1 as extendFreezingPeriod.
It is possible, because chest is still frozen
chest.increaseStake(
positionIndex,
amount*10,
1
);
vm.stopPrank();

// If succeed, then MIN_FREEZING_PERIOD was bypassed
vestingPosition = chest.getVestingPosition(
positionIndex
);

powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

// Our voting power is much bigger for short amount of time
```

See more

Results:

`MIN_FREEZING_PERIOD` was bypassed due to the fact, that chest was frozen. Our voting power, for short amount of time, increased drastically.

F-2024-1560 - claimWeeks does not burn tokens in StakingRewardDistribution - Medium

Description: For both `LiquidityRewardDistribution` and `StakingRewardDistribution` contracts, the logic implements two options for claiming rewards. One of them is claiming a single week rewards, second one, claiming multiple weeks rewards through the `claimWeeks` function.
In addition, the user has two options for claiming rewards:

- claim now and lose a 50 percentage of rewards
- start a 30 day reward vesting period to get 100% of rewards

It was found that in the `StakingRewardDistribution` contract, in the `claimWeeks` function when choosing immediate payout of rewards, the user actually receives only 50% of them, but there is no `burn` operation to destroy the other half. Consequently, these tokens remain locked in the contract and will not be deducted from `totalSupply`, which is the case in the three others rewards claiming functions in the protocol.

contracts/StakingRewardsDistribution.sol#198:

```
if (totalBalance > 0) {
if (_tokens[token] == jellyToken) {
if (_isVesting) {
_tokens[token].approve(vestingContract, totalBalance);
RewardVesting(vestingContract).vestStaking(
totalBalance,
msg.sender
);
} else
_tokens[token].safeTransfer(
msg.sender,
totalBalance / 2
);
} else _tokens[token].safeTransfer(msg.sender, totalBalance);
}
```

Assets:

- StakingRewardDistribution.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Fixed

Classification

Impact:	1/5
Likelihood:	5/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation: We suggest that half of the tokens be burned in the event of an immediate claiming of funds by the user in `StakingRewardDistribution`, as is the case in the `LiquidityRewardDistribution` contract.

Resolution: The Finding was fixed in commit `29fe12f` by using the `burn` function also for `StakingRewardsDistribution`.

F-2024-1509 - Voting power restored without cost after cliff period - Low

Description: Following the logic of the `Chest` contract, voting power declines over time. However, for regular `Chest`, if it hits cliff time, there is a bypass possible, to regain the full voting power.

If voting power will drop to zero, user needs to increase its stake by 1 to get all his voting power back. Then he doesn't need to pay a fee and he doesn't need to stake `min_stake_amount`.

Assets:

- Chest.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Accepted

Classification

Impact:	1/5
Likelihood:	2/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: The logic of the `Chest` contract should be adjusted so that a possible increase in stake by a small amount does not reset the entire voting power, but is calculated adequately to the current vesting status. After the cliff time has passed, the old stake should not be taken into account in the calculations.

Remediation (Accepted): This finding was acknowledged by the JellyVerse team with the comment:

> Voting power is restored if and only if the user extends the freezing period. In actuality voting power is restored by refreezing, not by adding tokens to staking, as can be seen in the reproducibility example. Such behavior is intended, as voting power should dependent on the freezing amount and time until unfreezing.

Evidences

Proof of Concept

Reproduce:

```
function test_regular_getChestPower_after_increaseStake_after_cliff() e
xternal {
vm.warp(1713695934);

uint256 amount = MIN_STAKING_AMOUNT * 100;
uint32 freezingPeriod = MIN_FREEZING_PERIOD * 10;

vm.startPrank(testAddress);
jellyToken.approve(address(chest), amount + chest.fee());s

// Stake is done here
chest.stake(amount, testAddress, freezingPeriod);

vm.stopPrank();

uint256 positionIndex = 0;
Chest.VestingPosition memory vestingPosition = chest.getVestingPosition
(
positionIndex
);

uint256 powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

// We should have whole voting power now
assertEq(1000, powerGetter);

// We are teleporting to the cliffTimestamp, where voting power should
be reduced to zero
vm.warp(
vestingPosition.cliffTimestamp
);

powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

// We are confirming 0 voting power
assertEq(0, powerGetter);

// We are increasing stake by one
vm.startPrank(testAddress);
jellyToken.approve(address(chest), 1);
chest.increaseStake(
positionIndex,
1,
freezingPeriod
);
vm.stopPrank();

vestingPosition = chest.getVestingPosition(
positionIndex
);

powerGetter = chest.estimateChestPower(
block.timestamp,
vestingPosition
);

// Whole voting power was recovered
assertEq(1064, powerGetter);
}
}
```

Results:

As a result, whole voting power was recovered just by staking one more token.

F-2024-1510 - Missing fee validation in setFee function - Low

Description:

Some operations, such as staking, have a mechanism for collecting a certain fee on the action performed, which is then available to the owner for withdrawal via the `withdrawFees` function in `Chest.sol`. The fee amount is also under the control of the contract owner and can be changed via the `setFee` function.

However, this function accepts the `fee_` parameter without any validation. It must only be in the `uint128` type range.

Due to the fact, that owner can specify any fee amount, it might be problematic for users, cause it opens an attack vector, if for example, owner keys are compromised.

Chest.sol#355:

```
function setFee(uint128 fee_) external onlyOwner {
fee = fee_;
emit SetFee(fee_);
}
```

Assets:

- Chest.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Accepted

Classification

Impact:	2/5
Likelihood:	1/5
Exploitability:	Dependent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation:

We recommend implementing validation of the `fee_` field in the form of introducing an upper fee limit that can be set. For a fairer fee collection, we also suggest considering collecting the percentage of the value against which the fee is calculated, unlike constant value.

Remediation (Accepted): This finding was acknowledged by the JellyVerse team with the comment:

> The owner of those contracts will be the governance contract, so setting fees can be done only through a governance proposal. As

such it is on the voting participants to make sure that the fee proposed is in line with the best interests of the ecosystem. The attack vector given in the example is not possible: the owner keys cannot be compromised, as the owner is a contract.

F-2024-1559 - Lack of upper bound Beneficiaries may lead to deny of service - Low

Description: It was noted that in the `Minter` contract, the owner of the contract may at any time replace the list of beneficiaries with a new one indicated by him. This is problematic because the contract does not validate the number of objects transferred under `Beneficiary[] calldata _beneficiaries`.

It is therefore possible that a very large number of them will be saved, which may result in a situation in which the call to the `mint` function will return an error due to too many operations performed and the gas limit being saturated within the `for` loop in `#129`.

contracts/Minter.sol:129

```
for (uint16 i = 0; i < beneficiaries.length; i++) {
uint256 weight = beneficiaries[i].weight;
uint256 amount = (mintAmountWithDecimals * weight) / 1000;


if (beneficiaries[i].beneficiary == stakingRewardsContract) {
IJellyToken(i_jellyToken).mint(address(this), amount);
IJellyToken(i_jellyToken).approve(
stakingRewardsContract,
amount
);
epochId = IStakingRewardDistribution(stakingRewardsContract)
.deposit(IERC20(i_jellyToken), amount);
} else {
IJellyToken(i_jellyToken).mint(
beneficiaries[i].beneficiary,
amount
);
}
}
```

contracts/Minter.sol:223:

```
function setBeneficiaries(
Beneficiary[] calldata _beneficiaries
) external onlyOwner {
delete beneficiaries;

uint256 size = _beneficiaries.length;
for (uint256 i = 0; i < size; ++i) {
beneficiaries.push(_beneficiaries[i]);
}
emit BeneficiariesChanged();
}
```

Assets:

- Minter.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Accepted

Classification

Impact: 1/5

Likelihood: 1/5

Exploitability:	Dependent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: We suggest introducing a safe, proven upper limit on the number of beneficiaries that can be saved in the contract state via the `setBeneficiaries` function.

Remediation (Accepted): This finding was acknowledged by the JellyVerse team with the comment:

> In an effort to not add additional constraints, we kept the implementation as is. It is on the voting participants to make sure that the number of beneficiaries is kept bellow a threshold that would result in a denial of service. If such a case does occur the denial of service would be temporary, and the issue can easily be resolve by reducing the number of beneficiaries to a manageable amount. In practical terms that number will always be in a range between 2 and 20, as the beneficiaries should not be individual users, but rather protocol specific distributor contract.

[F-2024-1563](F-2024-1563) - dailySnapshot can revert due to integer underflow - Low

Description:

It was noticed that one of the checks in the `dailySnapshot` function is to verify that the current timestamp is greater than `beginningOfTheNewDayTimestamp+ONE_DAY_SECONDS`. If so, the function execution continues. Otherwise, a `DailySnapshot_TooEarly` error is returned.

However, if the snapshot is executed, the `beginningOfTheNewDayTimestamp` parameter is incremented by `ONE_DAY_SECONDS`, i.e. the value of one day. If `dailySnapshot` is executed again when `beginningOfTheNewDayTimestamp` is greater than `block.timestamp`, an integer underflow will occur on line #69, which is an unexpected response of the contract.

contracts/DailySnapshot.sol:67:

```
function dailySnapshot() external onlyStarted {
if (
block.timestamp - beginningOfTheNewDayTimestamp <= ONE_DAY_SECONDS
) {
revert DailySnapshot_TooEarly();
}
uint48 randomBlockOffset = uint48(block.prevrandao) % oneDayBlocks;
uint48 randomDailyBlock = beginningOfTheNewDayBlocknumber +
randomBlockOffset;
dailySnapshotsPerEpoch[epoch][epochDaysIndex] = randomDailyBlock;

emit DailySnapshotAdded(
msg.sender,
epoch,
randomDailyBlock,
epochDaysIndex
);

unchecked {
beginningOfTheNewDayTimestamp += ONE_DAY_SECONDS;
beginningOfTheNewDayBlocknumber += oneDayBlocks;
++epochDaysIndex;
}
if (epochDaysIndex == 7) {
unchecked {
epochDaysIndex = 0;
++epoch;
}
}
}
```

Due to the fact that the consequence of this situation is revert of the transaction, the risk has been assessed as low.

Assets:

- DailySnapshot.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Fixed

Classification

Impact: 3/5

Likelihood:	1/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Low

Recommendations

Remediation: We recommend that you validate the `beginningOfTheNewDayTimestamp` parameter against `block.timestamp` to verify that `dailySnapshot` can execute properly. If it is larger, an appropriate error should be returned.

Resolution: The Finding was fixed in commit `437afae` by changing the order of mathematical operations, which prevents underflow from occurring.

Observation Details

F-2024-1564 - buyWithUsd will revert for USDT due to IERC20 approve - Info

Description:

The `buyWithUsd` function from the `PoolParty` contract is used to buy jelly tokens with USD pegged token. The technical documentation contains information that the `usdtoken` parameter will be intended for the DUSD token, but this is not directly stated in the contract itself.
In this regard, it should be noted, that on line `#123` an `approve` operation is performed on the `IERC20(usdtoken)` parameter. If `usdtoken` is assigned to an address of, for example, `USDT`, which is not compatible with the `IERC20` interface, this operation will return an error. This is due to the fact that `approve` returns a boolean value and the USDT interface returns a void value.We recommend taking this observation into account if, for some reason, the protocol decides not to use the DUSD token as the main USD pegged token in the `buyWithUsd()` function.

Consequently, it will be impossible to execute the `buyWithUsd` function with the `USDT` token.

contracts/PoolParty.sol#121:

```
//approve jelly tokens to be spent by jellySwapVault
IJellyToken(i_jellyToken).approve(jellySwapVault, jellyAmount);
IERC20(usdToken).approve(jellySwapVault, _amount);

IVault(jellySwapVault).joinPool(
jellySwapPoolId,
sender,
recipient,
request
);

IJellyToken(i_jellyToken).safeTransfer(msg.sender, jellyAmount);
```

Due to the fact that `DUSD` will probably be the only supported token here, this vulnerability is only considered as an observation. However, it should be kept it in mind if `usdtoken` is changed in the future.

Assets:

- PoolParty.sol [https://github.com/MVPWorkshop/jelly-verse-contracts]

Status: Accepted

Recommendations

Remediation: We recommend taking this observation into account if, for some reason, the protocol decides not to use the `DUSD` token as the main USD pegged token in the `buyWithUsd` function.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/MVPWorkshop/jelly-verse-contracts/
Commit	3331a1e0ff6def4aec7a9aa736b6e96b18140409
Whitepaper	https://hackenio.cc/hacken-methodologies
Requirements	https://github.com/MVPWorkshop/jelly-verse-contracts/docs/*
Technical Requirements	https://github.com/MVPWorkshop/jelly-verse-contracts/README.md

Contracts in Scope

contracts/Chest.sol

contracts/DailySnapshot.sol

contracts/Governor.sol

contracts/GovernorVotes.sol

contracts/InvestorDistribution.sol

contracts/JellyGovernor.sol

contracts/JellyTimelock.sol

contracts/JellyToken.sol

contracts/JellyTokenDeployer.sol

contracts/LiquidityRewardDistribution.sol

contracts/Minter.sol

contracts/OfficialPoolsRegister.sol

contracts/PoolParty.sol

contracts/RewardVesting.sol

contracts/StakingRewardDistribution.sol

contracts/TeamDistribution.sol

contracts/utils/Ownable.sol

contracts/utils/Vesting.sol

contracts/vendor/prb/SD59x18/Casting.sol

Contracts in Scope

contracts/vendor/prb/SD59x18/Constants.sol

contracts/vendor/prb/SD59x18/Conversions.sol

contracts/vendor/prb/SD59x18/Math.sol

contracts/vendor/prb/SD59x18/Errors.sol

contracts/vendor/prb/SD59x18/ValueType.sol

contracts/vendor/prb/SD59x18/Helpers.sol

contracts/vendor/balancer-labs/v2-interfaces/v0.4.0/pool-weighted/WeightedPoolUserData.sol

contracts/extensions/GovernorCountingSimple.sol

contracts/extensions/GovernorSettings.sol

contracts/extensions/GovernorTimelockControl.sol

contracts/interfaces/IChest.sol

contracts/interfaces/IJellyToken.sol

contracts/interfaces/IStakingRewardDistribution.sol