coinspect

You build, we defend.

Smart Contract Audit

Project 6022

Dec, 2024

Protocol 6022
Smart Contract Audit

Version: v241230 | Prepared for: Protocol 6022 | December 2024

Security Assessment

1. Executive Summary
2. Summary of Findings
    2.3 Solved issues & recommendations
3. Scope
4. Assessment
    4.1 Security assumptions
    4.2 Decentralization
    4.3 Testing
    4.4 Code quality
5. Detailed Findings

© Coinspect 2024

P6022-01 - New vaults don't receive reward allocation upon creation
P6022-02 - Rewards will get permanently locked after reinvesting
P6022-03 - Vaults receive incorrect or zero amount of rewards when harvesting
P6022-04 - Incompatible vault conditions render rewards unclaimable
P6022-05 - Creation of expired vaults lose funds and become unusable
P6022-06 - Low reward amounts might be lost and not be assigned to any vault
P6022-07 - Irrecoverable dust is spontaneously generated after a reward distribution
P6022-08 - Vaults receive less rewards due to precision loss in fee calculations
P6022-09 - Unused code
6. Disclaimer

© Coinspect 2024

1. Executive Summary

In **November, 2024**, **6022** engaged Coinspect to perform a Smart Contract Audit of Protocol 6022. The objective of the project was to evaluate the security of the smart contracts involved.

Protocol 6022 is a reward distribution protocol that allows users to create pools and vaults. Each Vault receives shares of a reward pool where the owner can decide to claim or reinvest.

	Solved	Caution Advised	Resolution Pending
High	3	0	0
Medium	3	0	0
Low	0	0	0
No Risk	3	0	0
Total	9	0	0

During this assessment, Coinspect identified the following issues: `P6022-01` shows the risks of not updating the reward accountancy when creating new vaults, `P6022-02` explains a scenario where rewards for a vault remain locked into the pool, `P6022-03` warns about a scenario where a vault can receive an incorrect amount of rewards, and `P6022-04` shows how non compatible conditions render rewards unclaimable. Then, `P6022-05` shows how vaults created in the future harm the pool. Lastly, `P6022-06` explains how vault could receive no allocations with non-zero reward amounts and `P6022-07` shows how reward dust is spontaneously accumulated inside each pool.

© Coinspect 2024

2. Summary of Findings

This section provides a concise overview of all the findings in the report grouped by remediation status and sorted by estimated total risk.

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
P6022-01	New vaults don't receive reward allocation upon creation	High
P6022-02	Rewards will get permanently locked after reinvesting	High
P6022-04	Incompatible vault conditions render rewards unclaimable	High
P6022-05	Creation of expired vaults lose funds and become unusable	Medium
P6022-06	Low reward amounts might be lost and not be assigned to any vault	Medium
P6022-07	Irrecoverable dust is spontaneously generated after a reward distribution	Medium
P6022-03	Vaults receive incorrect or zero amount of rewards when harvesting	None
P6022-08	Vaults receive less rewards due to precision loss in fee calculations	None
P6022-09	Unused code	None

© Coinspect 2024

3. Scope

The scope was set to be the repository at https://github.com/6022protocol/6022 at commit `f428dabd57462966a7c6b1a1e2a085e40b37787a`.

4. Assessment

The project features a modular rewarding system that allows users to create rewarding pools. Then, each user can create vaults for each rewarding pool. Vaults are expected to accrue rewards through the reward pool where they were created.

Users have to deposit a token (`ERC20` or `ERC721`) into a vault in order to start accumulating rewards. When a user creates a vault, they receive three Vault NFTs.

The rewarding system considers several rewarding scenarios that trigger reward harvesting or reinvestment depending on several conditions such as lock times, current timestamp and the amount of Vault NFTs held.

The rewarding system estimates the amount of collected rewards only for eligible pools. Eligible pools must have a lock time in the future, and the user must have a stake in the vault (an ongoing deposit). Coinspect reported several issues related to the rewarding system and accrual. Also, the rewarding mechanism is custom-made for this project and does not follow any well known and stress tested mechanisms (e.g. Masterchef, Synthetix). **Considering the overall risk of the reported issues, Coinspect strongly recommends reimplementing the reward system using the before-mentioned mechanisms as a reference.**

4.1 Security assumptions

For this security assessment Coinspect assumed that:

- Operative variables and parameters of the project are defined correctly
- Users interact with the system deploying pools using their factories, instead of deploying their own contracts.

4.2 Decentralization

The project uses a role based access control for the Controller smart contract. It restricts and controls several methods related to the privileges for adding new pools to the internal state. Also, functions to modify administrator privileges are included. Coinspect identified the presence of horizontal take-over risk since anyone holding the administrator role can remove and add new admins.

© Coinspect 2024

4.3 Testing

Coinspect observed that the testing suite **does not reflect adversarial scenarios**. Additionally, tests that thoroughly evaluate how rewards are allocated, updated and distributed are not included. The test suite mainly reproduces superficial scenarios **considering the event emission as the only success condition**, and does not take into account different usage scenarios. For example, cases with multiple vaults, with different wanted token types, lock times, that combine different operations (harvest, reinvest), among other sample scenarios.

It is **strongly recommended** to include more adversarial tests as well as success conditions before advancing to the production phase to increase the chance of detecting bugs.

4.4 Code quality

Coinspect observed that several variables and smart contracts include NatSpec. However, most public/external functions do not include it. It is recommended to add the missing documentation in those functions.

Additionally, the documentation link provided at the `README` is outdated and mentions functions that no longer exist.

© Coinspect 2024

5. Detailed Findings

P6022-01

New vaults don't receive reward allocation upon creation

Location

`core/contracts/RewardPool6022.sol#L80-L96`

Description

All vaults in the system receive incorrect reward calculations because the protocol tracks non-existent fees, directly impacting the reward distribution fairness. This occurs because the `collectedFees` state variable, which is crucial for reward calculations, is incremented even when no actual fees are collected.

The issue starts when users create a vault through `RewardPool6022::createVault`. The protocol first calculates the expected fees:

```
    uint256 _protocolTokenFees = (_backedValueProtocolToken / 100) *
FEES_PERCENT;

    if (_rewardablePools() > 0) {
        protocolToken.transferFrom(msg.sender, address(this),
_protocolTokenFees);
        _updateRewards(_protocolTokenFees);
    }
...
```

However, vaults are not rewardable by default. A vault only becomes rewardable when it meets three conditions:

1. It must have received deposits
2. Its `lockedUntil` timestamp must be greater than the current `block.timestamp`
3. Must not have made the withdrawal

For example, the first vault in a pool, neither condition is met at creation time. This causes the `_rewardablePools() > 0` check to return false, skipping the fee transfer entirely. Despite no actual fees being collected, the protocol still creates the vault and incorrectly records phantom fees in the accounting.

```
    Vault6022 vault = new Vault6022(
        msg.sender,
        _name,
        _lockedUntil,
        _wantedAmount,
        address(this),
        _wantedTokenAddress,
        _storageType
    );
    allVaults.push(address(vault));
    isVault[address(vault)] = true;
@>  collectedFees[address(vault)] += _protocolTokenFees;  // Increments
fees even when no transfer occurred
```

This accountancy error has system-wide implications because `collectedFees` is used to calculate rewards for all vaults in the pool. The phantom fees artificially inflate the total fee pool, causing early vaults to receive unearned rewards while diluting rewards for vaults that actually paid fees.

Recommendation

Handle reward updates so they consider newly created vaults.

© Coinspect 2024

Status

Fixed on commit `52127fbc83777ba61003fd7db5e72ce5ae3b7711`.

Vault creation was fixed to allocate token fees for all vaults.

Proof of Concept

The test will demonstrate that `collectedFees` shows a balance in the vault even when the actual token balance in the vault is 0:

```
collectedFees Balance: 2000000000000000000
protocolTokenBalance Balance:                              0
```

```
it("Coinspect - collected fees are greater than protocol token
balance", async function (){
      const { vault6022, rewardPool6022 } = await loadFixture(
        deployDepositedVaultFixture
        );

      // Get the protocol token
      const protocolTokenAddress = await
rewardPool6022.protocolToken();

      // Get collected fees for the vault
      const collectedFees = await rewardPool6022.collectedFees(await
vault6022.getAddress());

      // Create IERC20 interface for protocol token
      const protocolToken = await ethers.getContractAt("IERC20",
protocolTokenAddress);
      const protocolTokenBalance = await protocolToken.balanceOf(await
rewardPool6022.getAddress());

      console.log("collectedFees Balance: ", collectedFees );
      console.log("protocolTokenBalance Balance: ",
protocolTokenBalance );

      // Assert that collected fees are greater than protocol token
balance
      expect(collectedFees).to.be.gt(protocolTokenBalance);

    });
```

P6022-02

Rewards will get permanently locked after reinvesting

Location

`core/contracts/Vault6022.sol#L157-L162`

Description

Users permanently lose access to their rewards when using `ERC721` tokens as `wantedToken` due to a flawed withdrawal mechanism. This happens because the protocol attempts to transfer the same NFT token multiple times, even after it's already been withdrawn, causing all subsequent withdrawal transactions to revert.

The issue occurs in the following sequence:

1. User calls `RewardPool6022::createVault()` specifying an NFT contract as `wantedTokenAddress` with `storageType = VaultStorageEnum.ERC721`
2. User deposits the NFTs `wantedAmount` via `Vault6022::deposit()`
3. Vault accumulates rewards over time through protocol operations while `lockUntil` is less than `block.timestamp`
4. User performs initial withdrawal via `Vault6022::withdraw()` which:

© Coinspect 2024

- Transfers the NFT `wantedAmount` back to the user
- Sets `isWithdrawn = true`
- Reinvests existing rewards through `rewardPool.reinvestRewards()`. This step distributes rewards across all rewardable pools, diluting its amount.

When trying to claim the reinvested rewards through another `withdraw()` call, the function attempts to transfer the same `wantedAmount` again, but fails since the NFT is no longer in the vault's possession.

```
function withdraw() public nonReentrant {
    if (!isDeposited) {
        revert ContractNotDeposited();
    }

    uint256 requiredNFTs = getRequiredNftsToWithdraw();
    if (requiredNFTs > balanceOf(msg.sender)) {
        revert NotEnoughtNFTToWithdraw();
    }

    if (storageType == VaultStorageEnum.ERC721) {
        IERC721 wantedToken = IERC721(wantedTokenAddress);

        wantedToken.transferFrom(address(this), msg.sender,
wantedAmount);
        emit Withdrawn(msg.sender, wantedAmount);
    } else {
        IERC20 wantedToken = IERC20(wantedTokenAddress);
        uint256 balance = wantedToken.balanceOf(address(this));

        wantedToken.transfer(msg.sender, balance);
        emit Withdrawn(msg.sender, balance);
    }

    isWithdrawn = true;
    withdrawTimestamp = block.timestamp;

    if (requiredNFTs == WITHDRAW_NFTS_LATE) {
        rewardPool.harvestRewards(msg.sender);
    } else {
        rewardPool.reinvestRewards();
    }
}
```

The repeated transfer attempt causes two critical issues:

- For ERC721: The transaction reverts because the NFT is no longer in the vault, causing the rewards to remain locked. Even if the policyholder or Insurance attempts to deposit again the `wantedAmount` NFT to withdraw the rewards, they create a security risk, when `lockedUntil < block.timestamp`, any counterparty can withdraw both the `wantedAmount` and rewards since the required NFT holding check drops from 2 to 1.
- For ERC20: While the transaction succeeds, it wastes gas attempting a zero value transfer

© Coinspect 2024

Also, if the design intends that users can only call `withdraw()` once, redistributing their rewards across other vaults make other issues to arise:

1. Users are allowed to bypass single-withdrawals by directly transferring the wanted NFT to the Vault, and call `withdraw()` again.
2. The reward pool will still have a non-zero `collectedFees` value for the vault. As a consequence, the portion of rewards allocated to that vault will remain locked into the Reward Pool.

Recommendation

Redesign the withdrawal mechanism so it: ensures token availability (either NFT or ERC20) and uses updated values for accrued rewards.

The withdrawal mechanism should properly handle rewards at the time of withdrawal since the vault becomes non-rewardable afterwards (`isRewardable` returns false when `isWithdrawn` is true). Distribute pending rewards during withdrawal before marking the vault as withdrawn.

Status

Fixed on commit `52127fbc83777ba61003fd7db5e72ce5ae3b7711`.

Vaults now include modifiers that prevent multiple deposits and withdrawals, making this issue infeasible.

Proof of Concept

The following test demonstrates that rewards get permanently locked after reinvesting with ERC721:

```
NFT balance in vault before withdrawal: 1
NFT balance in vault after first withdrawal: 0

Attempting second Withdrawal:
// Second withdrawal fails because the NFT is no longer in the vault
NFT balance in vault after second withdrawal: 0
```

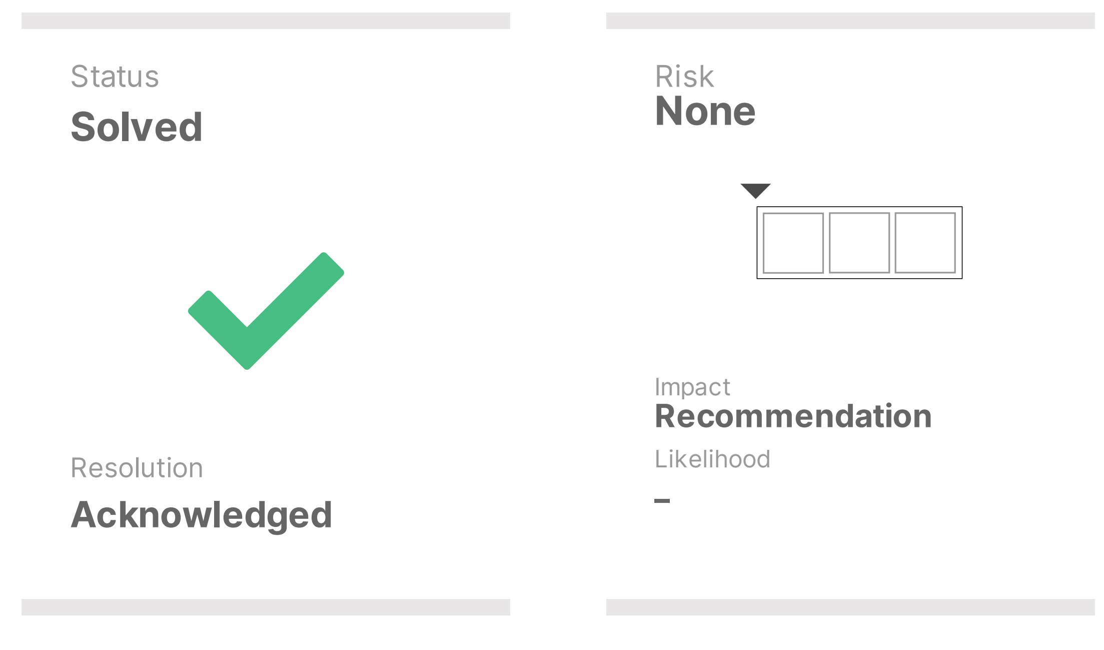
```
it("Coinspect - Rewards get locked after reinvesting with ERC721",
async function () {
    const { rewardPool6022, token6022, owner } = await loadFixture(
        deployEmptyVaultFixture
    );
```

© Coinspect 2024

```
    // Setup: Create vault with ERC721 token and deposit
    const MockERC721 = await ethers.getContractFactory("MockERC721");
    const mockNFT = await MockERC721.deploy("MockNFT", "MNFT");

    // Mint NFT to owner
    await mockNFT.mint(await owner.getAddress(), 1);

    // Create new vault using ERC721
    const tx = await rewardPool6022.connect(owner).createVault(
        "Vault6022",
        lockUntil,
        1, // wantedAmount is 1 NFT
        await mockNFT.getAddress(),
        1, // VaultStorageEnum.ERC721
        ethers.parseEther("10")
    );

    const txReceipt = await tx.wait();
    const events = txReceipt?.logs.filter((x) => x instanceof EventLog)
as EventLog[];
    const vaultCreatedEvent = events.filter(
        (x) => x.fragment.name === "VaultCreated"
    )[0];

    const nftVault = await ethers.getContractAt("Vault6022",
vaultCreatedEvent.args[0]);

    // Approve and deposit NFT
    await mockNFT.connect(owner).approve(await nftVault.getAddress(),
1);
    await nftVault.connect(owner).deposit();

    console.log(`NFT balance in vault before withdrawal: ${await
mockNFT.balanceOf((await nftVault.getAddress()))}`);

    // First withdrawal
    await nftVault.connect(owner).withdraw();

    console.log(`NFT balance in vault after first withdrawal: ${await
mockNFT.balanceOf((await nftVault.getAddress()))}`);

    // Try to withdraw reinvested rewards but it was reverted
    console.log("\nAttempting second Withdrawal:");
    await expect(
        nftVault.connect(owner).withdraw()
    ).to.be.reverted;

});
```

P6022-03

Vaults receive incorrect or zero amount of rewards when harvesting



Location

`core/contracts/RewardPool6022.sol#L103`

Description

Vaults performing late withdrawals in order to harvest rewards will not receive the correct amount of rewards, ranging from receiving zero rewards to all the available rewards in the pool. This happens because reward harvesting does not update the collected rewards before claiming.

When a vault harvest rewards, `RewardPool.harvestRewards()` is called:

```
function harvestRewards(address to) external onlyVault {
    uint256 valueToHarvest = collectedRewards[msg.sender];
    collectedRewards[msg.sender] = 0;

    protocolToken.transfer(to, valueToHarvest);
```

© Coinspect 2024

```
        emit Harvested(msg.sender);
    }
```

This function transfers a `valueToHarvest` amount to the recipient. However, this parameter is only updated under two circumstances:

- When a new vault is created AND fees are distributed.
- When any vault decides to reinvest.

The vault creation process only updates the rewards' state only if there are other rewardable vaults in the system (see **P6022-01**). Because of this, the reward state is lagged and does not consider the shares of at least one vault.

A vault is considered rewardable after meeting the following conditions:

```
    function isRewardable() external view returns (bool) {
        return lockedUntil > block.timestamp && isDeposited &&
!isWithdrawn;
    }
```

Also, reward updates are not triggered if no vault decides to reinvest.

```
   function _updateRewards(uint256 amount) internal {
        uint256 totalRewardableVaults = _rewardablePools();

        if (totalRewardableVaults == 0) return; // No vaults to reward
and avoid division by zero

        for (uint i = 0; i < allVaults.length; i++) {
            Vault6022 vault = Vault6022(allVaults[i]);
            if (vault.isRewardable()) {
                // If there is only one vault, it will get all the past
rewards
                if (totalRewardableVaults == 1) {
                    collectedRewards[address(vault)] =
protocolToken.balanceOf(address(this));
                } else {
                    collectedRewards[address(vault)] += amount *
collectedFees[address(vault)] / totalRewardableVaults;
                }
            }
        }
    }
```

Consider the following scenario, a Reward Pool with only a single vault is created. When this vault is created, there are no rewardable vaults. Meaning that `_updateRewards()` is never called. Then, the vault owner decides to harvest rewards after depositing. Calls `withdraw()` after the lock period. No rewards are transferred to that vault.

Also, a similar scenario with two vaults, allows the first vault to steal all rewards since the vault creation assumes that there is only one rewardable

vault. When creating the second vault, if only the first one is rewardable, all the Pool's balance will be assigned to that vault. Then, if no user decides to reinvest the rewards, `_updateRewards()` will not be invoked and the first vault will get all the rewards.

Coinspect considers this issue to have no risk since rewards are only injected into the system when reinvesting or creating new vaults, and the smart contract updates the reward state when performing those actions. However, this scenario should be considered if the code is reused in a different context where rewards are deposited/withdrawn to the system.

Recommendation

Handle reward updates when harvesting rewards.

Status

Acknowledged.

The P6022 Team stated that policyholders are only rewarded by fees paid by the insurer on new policyholders.

Proof of Concept

The following test only logs the collected rewards by a vault that harvests. It can be seen that no rewards are sent to the claimer:

```
Rewards to be collected by Vault: 0
```

```
it("Coinspect - Harvest zero rewards due to lack of update", async
function () {
    const { vault6022, rewardPool6022, otherAccount } = await
loadFixture(
    deployDepositedVaultFixture
    );

    await time.increase(lockIn);

    console.log(`Rewards to be collected by Vault: ${await
rewardPool6022.collectedRewards(vault6022.getAddress())}`)
    await expect(vault6022.connect(otherAccount).withdraw())
    .to.emit(vault6022, "Withdrawn")
```

```
    .to.emit(rewardPool6022, "Harvested");
});
```

© Coinspect 2024

P6022-04

Incompatible vault conditions render rewards unclaimable

Location

`core/contracts/Vault6022.sol#L180`
`core/contracts/Vault6022.sol#L184`

Description

Users are allowed to harvest rewards only after the `lockUntil` time, however, no rewards will be accounted for that period since the vault is marked as non-rewardable.

This happens because the two states don't overlap and are exclusive. When withdrawing, rewards are harvested only after the `lockedUntil` time.

```
function getRequiredNftsToWithdraw() public view returns (uint256) {
    return block.timestamp < lockedUntil ? WITHDRAW_NFTS_EARLY :
WITHDRAW_NFTS_LATE;
}
```

© Coinspect 2024

This condition is required to trigger reward harvesting, that only occurs when the required NFTs matches the `WITHDRAW_NFTS_LATE`:

```
isWithdrawn = true;
withdrawTimestamp = block.timestamp;

if (requiredNFTs == WITHDRAW_NFTS_LATE) {
    rewardPool.harvestRewards(msg.sender);
} else {
    rewardPool.reinvestRewards();
}
```

However, the reward pool only considers rewardable pools as those who match the following conditions:

1. `lockedUntil > block.timestamp`
2. `isDeposited == true`
3. `isWithdrawn == false`

```
function isRewardable() external view returns (bool) {
    return lockedUntil > block.timestamp && isDeposited &&
!isWithdrawn;
}
```

This means that when reaching the harvesting or reinvestment point, no vault will be considered rewardable and thus, the Reward Pool will not be able to update their accrued rewards.

Regarding reward harvesting, `lockedUntil` will be in the past and `isWithdrawn` will be true (conditions 1 and 3 are not met). Then for reward reinvestments, `isWithdrawn` will be true (condition 3 is not met).

Coinspect considers this issue a concern because the rewarding system only updates the state of rewardable pools, in addition to `P6022-01`, `P6022-02` and `P6022-03`.

Recommendation

Redesign the claimable conditions to ensure that the reward's state for vaults is updated.

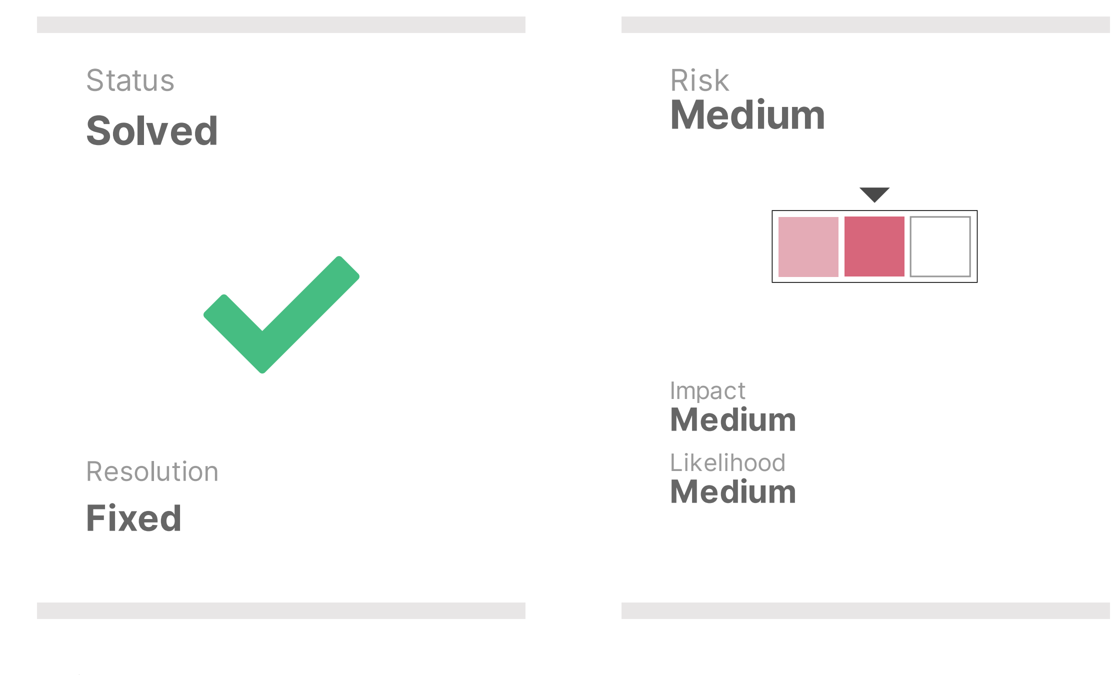
Status

Fixed on commit `52127fbc83777ba61003fd7db5e72ce5ae3b7711`.

Withdraw can now be called only once per vault.

P6022-05

Creation of expired vaults lose funds and become unusable



Location

`core/contracts/Vault6022.sol#L112`

Description

Protocol fees transferred during vault creation become permanently locked if the vault is created with an expired or nearly expired `_lockedUntil` timestamp. This occurs because the protocol fails to validate the lock time during creation but enforces it during deposit, making the vault unusable while retaining any transferred fees.

This issue is manifested in through this sequence:

First, a user creates a vault with expired/nearly expired `_lockedUntil`:

```
function createVault(
    string memory _name,
    uint256 _lockedUntil,   // No validation of timestamp
    uint256 _wantedAmount,
```

```
    address _wantedTokenAddress,
    VaultStorageEnum _storageType
) external {
    // Protocol fees are transferred before timestamp validation
    if (_rewardablePools() > 0) {
        protocolToken.transferFrom(msg.sender, address(this),
_protocolTokenFees);
        _updateRewards(_protocolTokenFees);
    }

    // Vault created with potentially expired lock time
    Vault6022 vault = new Vault6022(
        msg.sender,
        _name,
        _lockedUntil,
        _wantedAmount,
        address(this),
        _wantedTokenAddress,
        _storageType
    );
}
```

Any attempt to deposit will revert due to expired lock time:

```
function deposit() public nonReentrant {
    // ... other checks ...

    if (block.timestamp > lockedUntil) {
        revert TooLateToDeposit();  // Always reverts if created with
expired time
    }
    // ... deposit logic never reached ...
}
```

Coinspect identified that this issue harms users as it follows:

1. Protocol fees already transferred are permanently locked into the vault
2. Vault becomes an unusable contract
3. Gas spent on deployment is wasted
4. No recovery mechanism exists for locked fees
5. Total fees are diluted by the non-claimable position

Recommendation

Ensure that `_lockedUntil` is validated to be greater than `block.timestamp` during vault creation, and implement a minimum deposit window.

Status

© Coinspect 2024

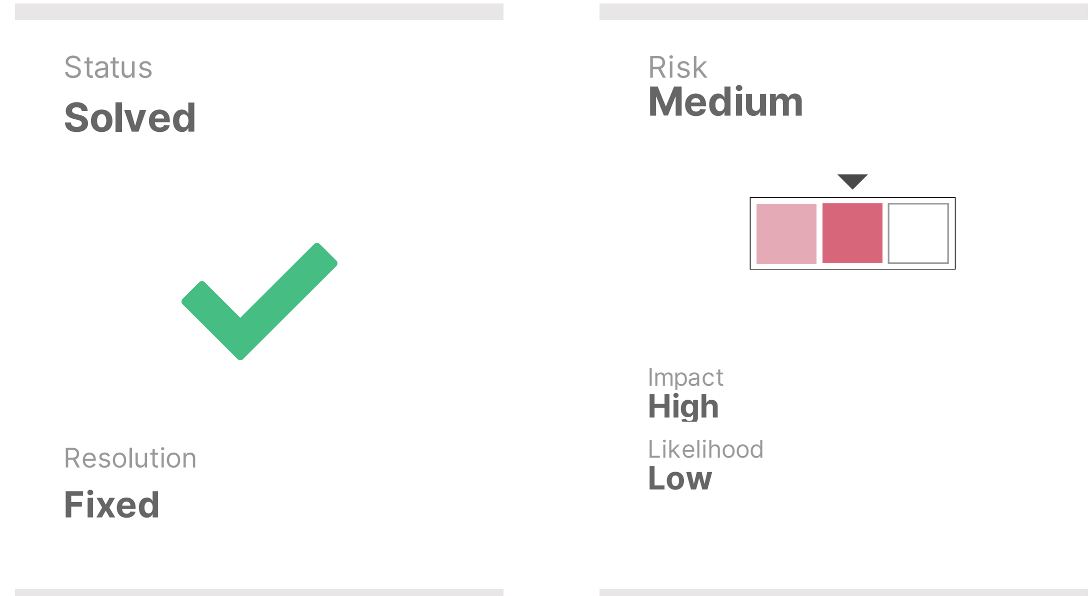
Fixed on commit `d95a1c2f40b02e86d7f1226f1346d0f92b631096`.

A minimum lock time was added.

© Coinspect 2024

P6022-06

Low reward amounts might be lost and not be assigned to any vault



Location

```
core/contracts/RewardPool6022.sol#L121
core/contracts/RewardPool6022.sol#L139
```

Description

Users could receive zero reward allocation even when they have non-zero `collectedFees`, since the reward allocation process does not require a minimum amount to distribute.

When rewards are updated, each vault receives an allocation proportional to the amount of shares (`collectedFees`) they own:

```
    function _updateRewards(uint256 amount) internal {
        uint256 totalRewardableVaults = _rewardablePools();

        if (totalRewardableVaults == 0) return; // No vaults to reward
and avoid division by zero
```

```
        for (uint i = 0; i < allVaults.length; i++) {
            Vault6022 vault = Vault6022(allVaults[i]);
            if (vault.isRewardable()) {
                // If there is only one vault, it will get all the past
rewards
                if (totalRewardableVaults == 1) {
                    collectedRewards[address(vault)] =
protocolToken.balanceOf(address(this));
                } else {
                    collectedRewards[address(vault)] += amount *
collectedFees[address(vault)] / totalRewardableVaults;
                }
            }
        }
    }
```

For small amounts to distribute or Reward Pools with multiple Vaults, the calculation for `collectedRewards` could be zero, as Solidity floors down divisions. Consider the following scenario:

- 100 vaults, `collectedFees` per vault is `1` (1% share)
- `amount = 10`

When distributing:

```
10 * 1 / 100 == 10 / 100 == 0
```

Recommendation

Consider including a minimum amount to distribute so the vault with the lowest amount of shares still receives at least one reward unit.

Status

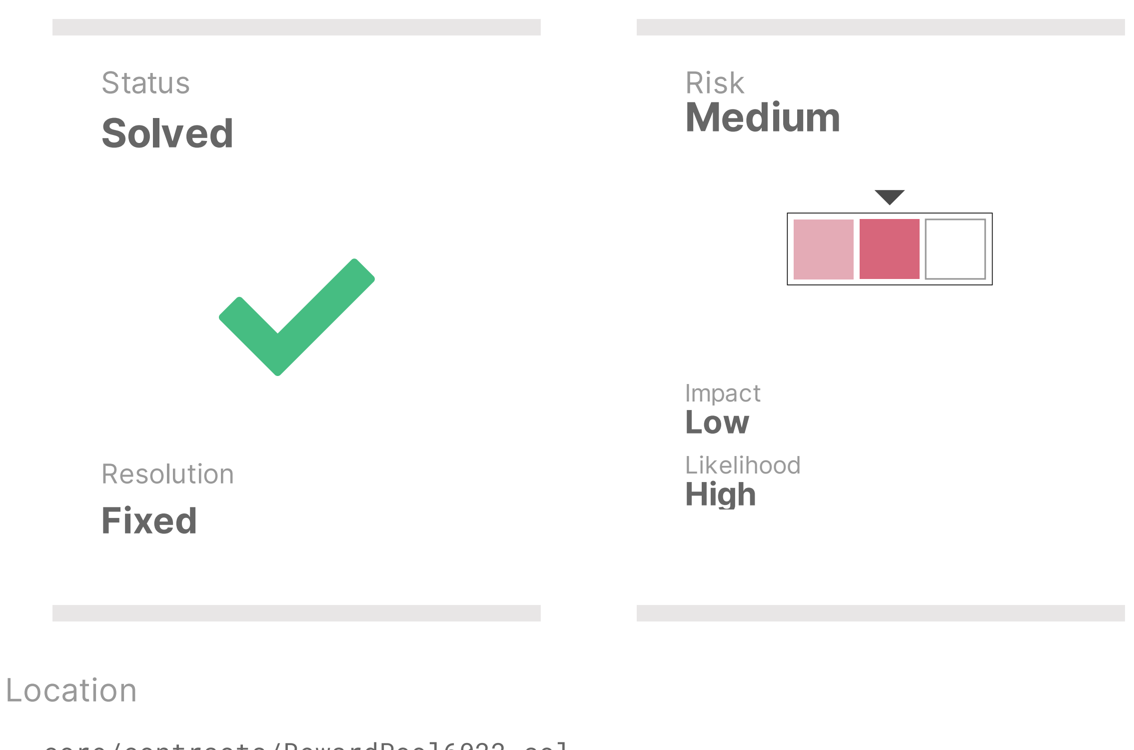
Fixed on commits `779f81d5b28efb87fd97c6b3b0d3d29b1b98e285` and `4b4adb1bc7acadd5d0f9ef546f6728284d51533a`.

Older vaults with low weight receive a slight bonus to prevent leaving dust.

© Coinspect 2024

P6022-07

Irrecoverable dust is spontaneously generated after a reward distribution



Location

core/contracts/RewardPool6022.sol

Description

Reward distributions might leave irrecoverable dust inside each Reward Pool, since the allocation does not consider divisions that have a remainder. As a consequence, a dust of rewards sent to the contract will remain locked.

Rewards are sent when creating a new vault, and it is expected that each vault claims them when harvesting. The distribution process might leave dust, as Solidity floors down the result of a division:

```
} else {
    collectedRewards[address(vault)] += amount *
collectedFees[address(vault)] / totalRewardableVaults;
}
```

© Coinspect 2024

Consider the following scenario: there are `1000` tokens to distribute, across two vaults (with `collectedFees` equal to 50 and 53 respectively). When calculating: the first vault receives `485.53 = 485 + 0.53 (dust, lost)` and the last vault receives `514.16 = 514 + 0.16 (dust, lost)`. A total of `1` unit is lost as dust only considering two vaults and this amounts. This value can increase considerably if multiple distributions are sequentially made.

Recommendation

Add a permissioned function to recover rewards dust.

Status

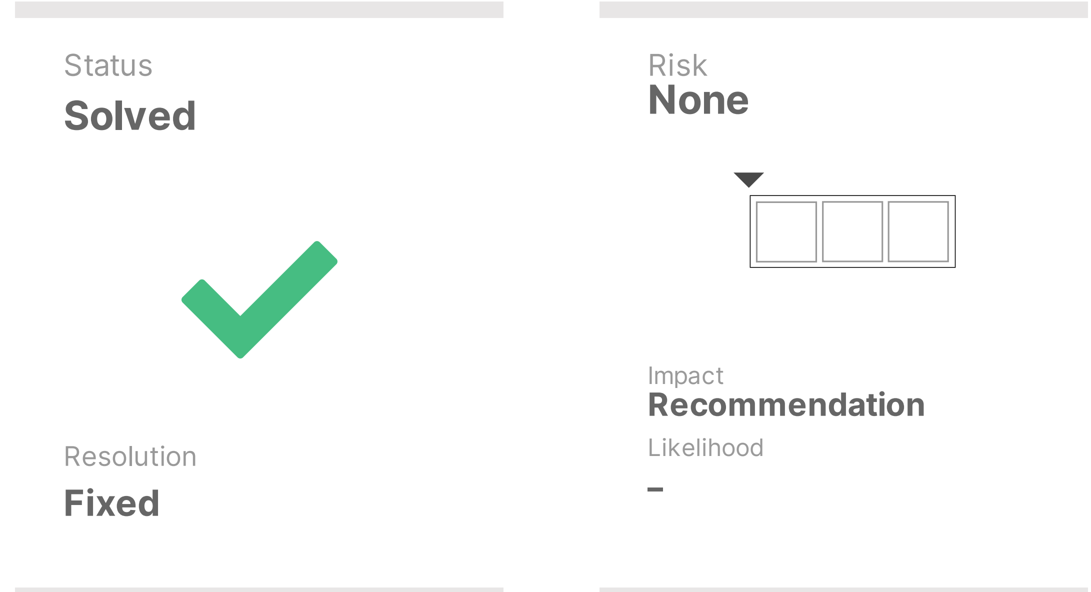
Fixed on commit `5a5bcf0d6c47f444cc323a738dfaa0b3a477fe2a`.

A function to recover dust was added. This function allows recovering remainders once the lifetime vault is no longer rewardable.

© Coinspect 2024

P6022-08

Vaults receive less rewards due to precision loss in fee calculations



Location

```
core/contracts/Vault6022.sol#L226
core/contracts/RewardPool6022.sol#L78
```

Description

Vaults receive fewer rewards than they should due to precision loss in the initial fee calculations. This issue compounds because collected fees directly determine reward distribution, and becomes particularly significant for tokens with low decimals like Gemini USD (2 decimals).

The precision loss begins during vault creation in `RewardProtocol6022::createVault`:

```
function createVault(...) external {
    // Fee calculation with precision loss that affects future rewards
    uint256 _protocolTokenFees = (_backedValueProtocolToken / 100) *
FEES_PERCENT;
```

© Coinspect 2024

```
    if (_rewardablePools() > 0) {
        protocolToken.transferFrom(msg.sender, address(this),
_protocolTokenFees);
        _updateRewards(_protocolTokenFees); // Reduced fees lead to
reduced rewards
    }
    // ...
}
```

Single Vault Example (using Gemini USD):

```
//Backed Value: $200.99 (20099 in token units)

// Current Implementation:
_protocolTokenFees = (20099 / 100) * 2
                   = 200 * 2
                   = 400 ($4.00)

// Correct Calculation:
_protocolTokenFees = (20099 * 2) / 100
                   = 40198 / 100
                   = 401 ($4.01)

// Lost rewards per vault: $0.01 worth of rewards
```

- Each affected vault has lower `collectedFees` recorded
- This reduces their share in `_updateRewards()` calculations

At 10,000 vaults: $100 GUSD worth of rewards are misallocated.

Recommendation

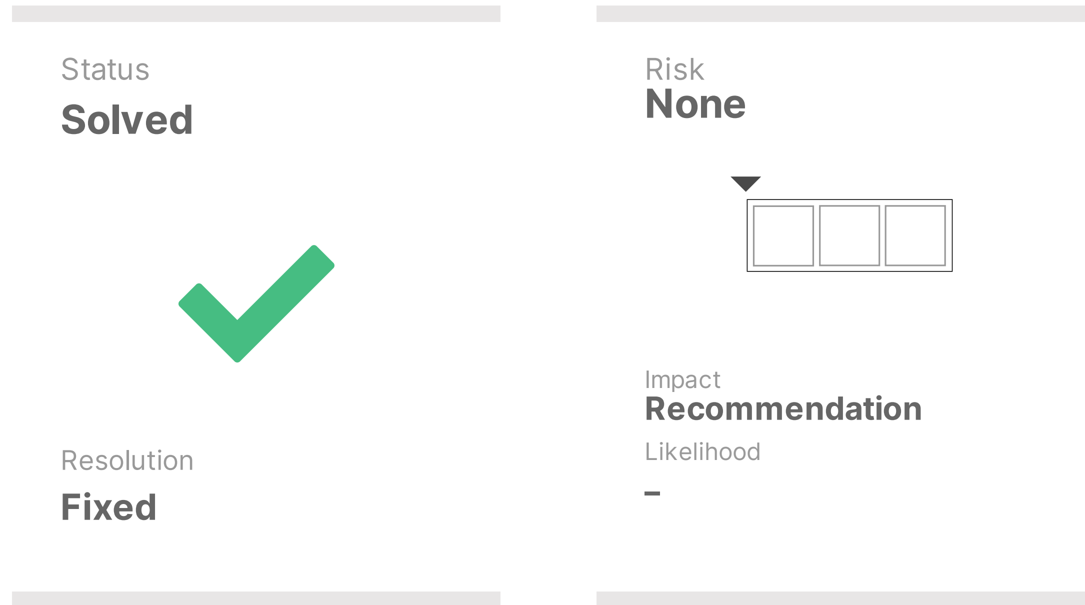
Perform multiplication before division to minimize precision loss and implement higher precision for fee calculation.

Status

Fixed on commit `689cc1c0f01add3defaedf64a9d7b42c4de71f5b`.

P6022-09

Unused code



Location

```
core/contracts/interfaces/IVault6022.sol#L7
core/contracts/interfaces/IController6022.sol#L4
core/contracts/Controller6022.sol#L4
core/contracts/interfaces/IRewardPoolFactory6022.sol
```

Description

The code contains several unused imports and an unimplemented interface:

1. Unused IERC20 import in multiple files:

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

Found in:

- `IVault6022.sol`
- `IController6022.sol`

2. Unused `IVault6022` import in Controller:

```
import {IVault6022} from "./interfaces/IVault6022.sol";
```

Found in:

- `Controller6022.sol`

3. Unimplemented empty interface inherited by `RewardPool6022`:

```
interface IRewardPoolFactory6022 {

}
```

Found in:

- `IRewardPoolFactory6022.sol`

Recommendation

Remove the unused imports from their respective files and implement the `IRewardPoolFactory6022` interface.

Status

Fixed on commit `52127fbc83777ba61003fd7db5e72ce5ae3b7711`.

© Coinspect 2024

6. Disclaimer

The contents of this report are provided "as is" without warranty of any kind. Coinspect is not responsible for any consequences of using the information contained herein.

This report represents a point-in-time and time-boxed evaluation conducted within a specific timeframe and scope agreed upon with the client. The assessment's findings and recommendations are based on the information, source code, and systems access provided by the client during the review period.

The assessment's findings should not be considered an exhaustive list of all potential security issues. This report does not cover out-of-scope components that may interact with the analyzed system, nor does it assess the operational security of the organization that developed and deployed the system.

This report does not imply ongoing security monitoring or guaranteeing the current security status of the assessed system. Due to the dynamic nature of information security threats, new vulnerabilities may emerge after the assessment period.

This report should not be considered an endorsement or disapproval of any project or team. It does not provide investment advice and should not be used to make investment decisions.

© Coinspect 2024