Aspecta AI Audit Report

HASHDIT

Audit Period: 2024/09/09 - 2024/09/23(YYYY/MM/DD)

Based on these links:

Project Name	Aspecta AI
Website	https://aspecta.ai
Audit Scope	https://github.com/aspecta-ai/contract-v1-core/tree/main-redeem/projects/asset-key-market

Disclaimer

1. The analysis of the Severity is purely based on the smart contracts mentioned in the Audit Scope and does not include any other potential contracts deployed by the Owner. No applications or operations were reviewed for Severity. No product code has been reviewed.
2. Due to the time limit, the audit team did not do much in-depth research on the business logic of the project. It is more about discovering issues in the smart contracts themselves.

Summary of Findings

ID	Title	Severity	Status
[High-01]	Reentrancy when selling key tokens allow draining of funds	High	Resolved
[High-02]	Project address can redeem aspecta assets free of charge by performing self redemptions	High	Resolved
[Medium-01]	Precision loss will leak key token value for project address during redemptions	Medium	Acknowledged
[Informational-01]	Admin functions could implement input validation and/or thresholds	Informational	Acknowledged

[High-01] Reentrancy when selling key tokens allow draining of funds

Description

Users can buy and sell key tokens within the AspectalKeyPools. Token purchase is performed through `buy()`, in which the caller can be a smart contract integrating the AspectaKeyPool contract.
Subsequently, as token supply increases with more purchases of tokens, the value of the tokens increases, and so users can sell the tokens whenever to realize profits. This is performed through `sell()`. Notice in the internal `_sell()` function, the following steps is performed:

1. Check that amount to sell is lesser or equal to seller's balance, if not revert
2. Compute current total price in BNB corresponding to amount to sell via `_calculateSaleReturn`
3. Check that totalPrice is lesser than minimum price
4. Compute project and protocol fee (assumed to be 2% each), and send to project and protocol respectively via `_sendFunds()`
5. Send net BNB to seller that has already accounted for fees back to seller via `_sendFunds()`
6. Burn sellers key token i.e. reduce totalySupply and sellers key token balance
7. Compute currentPrice after token sale via `_calculateSaleReturn` and emit a relevant event

```
function _sell(address seller, uint256 amount, uint256 minPrice) internal {
    uint256 totalPrice;
    uint256 projectFee;
    uint256 protocolFee;

    if (amount > _balances[seller]) {
        revert InsufficientKeys();
    }

    (totalPrice, _currentPiecewiseIndex, _currentC) = _calculateSaleReturn(
```

```
        amount
    );

    if (totalPrice < minPrice) {
        revert MinPriceNotMet(minPrice, totalPrice);
    }

    projectFee = (totalPrice * projectFeePercentage) / 1 ether;
    protocolFee = (totalPrice * protocolFeePercentage) / 1 ether;

    // Transfer fees
    _sendFunds(payable(projectAddress), projectFee);
    _sendFunds(payable(protocolFeeDestination), protocolFee);

    // Transfer funds back to the seller
    _sendFunds(payable(seller), totalPrice - projectFee - protocolFee);

    // Burn keys
    _burn(seller, amount);

    // Calculate the current buy price
    (uint256 currentBuyPrice, , ) = _calculatePurchaseReturn(1);
    emit Sell(
        projectAddress,
        seller,
        amount,
        totalPrice,
        projectFee,
        protocolFee,
        _balances[seller],
        _totalSupply,
        currentBuyPrice
    );
```

```
}
```

Notice how in steps 5 and 6, the sale proceeds in BNB are first sent to the seller via `_sendFunds()` utilizing a low level call, before burning the token balance.

```
function _sendFunds(address payable recipient, uint256 value) internal {
  (bool sent, ) = recipient.call{value: value}("");
  if (!sent) {
     revert SendFailure();
  }
}
```

Since the `seller` address is user controlled, this can be a smart contract having a malicious `fallback()` function that can call `sell()` again and again to repeatedly reenter the contract until right before the `minPrice` is hit. Again since the sales proceed is first sent before user token balance is subtracted, this will allow bypass of the `amount > balances[seller]` check, and as long as minPrice is not hit yet, it also bypasses the `totalPrice < minPrice` check, allowing each call to drain `totalPrice - projectFee - protocolFee` each time.
Note that in `buy()`, reentrancy is possible as well by utilizing the refund mechanism.
This causes the following impacts:

1. Complete draining of BNB within the AspectaKeyPools
2. Manipulation of key token prices since the `_totalSupply` variable will be overestimated/underestimated
3. Allow premature/late redemption of aspecta assets since `maxPiecewiseIndex` will be updated prematurely

Recommendation

1. Add reentrancy protection via the `nonReentrant` modifier
2. Always call `_burn/_mint` before sending funds to user via `_sendFunds()`

Fix Review

Fixed, both recommendation was implemented, namely

1. `nonReentrant` modifier is added to the internal `_buy()` and `_sell()` function.

2. In `_buy()`, `_mint()` is called before a refund is performed to the user. In `_sell()`, `_burn` is called before low level call to send native funds to user

[High-02] Project address can redeem aspecta assets free of charge by performing self redemptions

Description

The `projectAddress` is not a trusted entity, as evident by the amount of key tokens allowed to be sold being dictated by the allowance set by the `owner` via the `_spendProjectAllowance()`.

```
function sell(uint256 amount, uint256 minPrice) external whenNotPaused {
    _checkValidAmount(amount);
    _checkPoolActivated();
    if (msg.sender == projectAddress) {
@>      _spendProjectAllowance(amount);
    }
    _sell(msg.sender, amount, minPrice);
}
```

Project assets inherently hold values, and even though this assets value are determined by the project, we should be providing an avenue for the projectAddress to directly obtain this assets arbitrarily. However, this is possible due to the logic in `redeem()`. Notice for a whitelisted aspecta asset by the aspecta protocol admin, once redemption is allowed, the key tokens held by users will be transferred to the `projectAddress` to redeem the aspecta asset via `_transfer()`.

```
function redeem(
  uint256 amount,
  uint256 assetItemId
)
  external
  whenNotPaused
  returns (uint256 assetAmount, uint256 consumedKeyAmount)
```

```
{
    _checkValidAmount(amount);
    if (!assetItems.contains(assetItemId)) {
        revert AssetNotFound();
    }
    (assetAmount, consumedKeyAmount) = IAspectaAsset(
        assetItems.get(assetItemId)
    ).redeem(msg.sender, amount);
@>  _transfer(msg.sender, projectAddress, consumedKeyAmount);
    emit Redeem(
        projectAddress,
        msg.sender,
        assetItemId,
        consumedKeyAmount,
        assetAmount
    );
}
```

If the caller is the `projectAddress` itself calling redemption, this allows unlimited minting of the aspecta asset free or charge, as the internal _transfer function allows self transfers, so the amount subtracted will be offset by the amount added, essentially meaning no key tokens are deducted.

```
function _transfer(address from, address to, uint256 amount) internal {
    _balances[from] -= amount;
    _balances[to] += amount;
}
```

Recommendation

```
function redeem(
    uint256 amount,
    uint256 assetItemId
```

```
)
    external
    whenNotPaused
    returns (uint256 assetAmount, uint256 consumedKeyAmount)
{
+   require(msg.sender != projectAddress, "Caller should not be projectAddress");
    _checkValidAmount(amount);
    if (!assetItems.contains(assetItemId)) {
        revert AssetNotFound();
    }
    (assetAmount, consumedKeyAmount) = IAspectaAsset(
        assetItems.get(assetItemId)
    ).redeem(msg.sender, amount);
    _transfer(msg.sender, projectAddress, consumedKeyAmount);
    emit Redeem(
        projectAddress,
        msg.sender,
        assetItemId,
        consumedKeyAmount,
        assetAmount
    );
}
```

Fix Review

Fixed as recommended, `projectAddress` can no longer abitrarily redeem aspecta assets.

[Medium-01] Precision loss will leak key token value for project address during redemptions

Description

Once assets milestones are reached, users can perform keyToken redemptions to transfer keyTokens to project address in exchange for aspectaAssets, provided the asset is whitelisted and redemptions are unlocked by aspecta protocol admins. This is performed through the `redeem()` function.
Notice that this function in turn calls the `AspectaAsset.redeem()` function, which will compute the amount of key tokens to transfer to `projectAddress`, represented by `consumedKeyAmount` (`totalPrice` within the `AspectaAsset.redeem()` function)

- Notice how the `assetAmount` to mint to user is computed as `keyAmount / _price`
- Notice how the `totalPrice`, representing the keyToken to send to `projectAddress` for redemption of aspecta asset is computed as `assetAmount * _price`

This essentially means that it is expected that `totalPrice == keyAmount`, and as such the final computation is actually not required. However, since it was performed, the potential precision loss/rounding if `keyAmount` is not a multiple of `_price` will leak value and cost loss to projectAddress.

```
function redeem(
  address recipient,
  uint256 keyAmount
) external returns (uint256 assetAmount, uint256 totalPrice) {
  _checkPermission();

  if (msg.sender != address(aspectaKeyPool)) {
      revert InvalidCaller();
  }

@>  assetAmount = keyAmount / _price;
  if (assetAmount == 0) {
      revert InsufficientKeyInput();
  }

  uint256 totalMinted = _totalMinted() + assetAmount;
  if (totalMinted > _totalSupply) {
      revert NotEnoughTokensLeft(totalMinted, _totalSupply);
```

```
  }

  _safeMint(recipient, assetAmount);

@>   totalPrice = assetAmount * _price;
}
```

Recommendation

Remove the `totalPrice` variable in `AspectaAsset.redeem()` and directly utilize `amount` when calling `_transfer` within `AspectaKeyPool.redeem()`

Fix Review

Acknowledged as a design decision, only an integer amount of key tokens can be redeemed.

[Informational-01] Admin functions could implement input validation and/or thresholds

Description

The following permissioned function could implement relevant input validation and/or threshold checks.

AspectaAsset.sol

- `setStartTimestamp`: Could check `_newStartTimestamp` must be greater than current `block.timestamp`
- `setEndTimestamp`: Could check `_newEndTimestamp` must be greater than `_startTimestamp`
- `setKeyPoolAddress`: Could check `_newKeyPoolAddress` must be a non-zero address
- `setTotalSupply`: Could check `_newTotalSupply` must be a nonzero value. Could check that `_newTotalSupply` must be greater than current `totalMinted()`

AspectaKeyPool.sol

- `setUnlockTime`: Could check that `_unlockTime` must be greater than current `block.timestamp`
- `addAsset/removeAsset`: Could check that `assetItemAddress` must be a non-zero address

AspectaKeyPoolFactory.sol

- `createPool`: `projectFeePercentage` and `protocolFeePercentage` could be checked similar to `setProjectFeePercentage` and `setProtocolFeePercentage` respectively. `projectAddress` could be checked to be a non-zero address.

Fix Review

Acknowledged, no adjustments made but not an issue as long as admin sets an appropriate value.