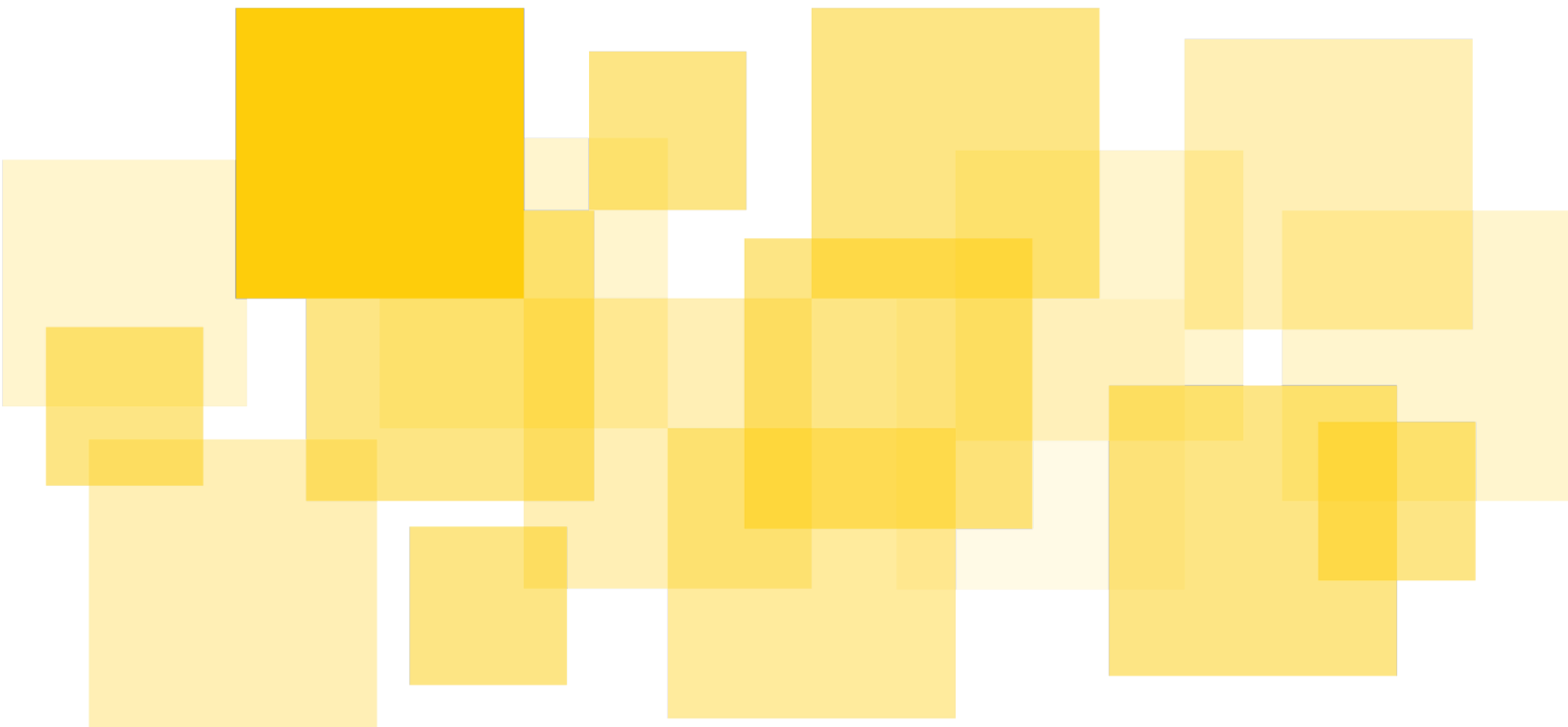


Audit Report

Gigastar Channel Revenue Distribution

Delivered: 2023-02-23



Prepared for GigaStar Technologies by Runtime Verification, Inc.





[Summary](#)

[Disclaimer](#)

[Channel Revenue Distribution: Contract Description and Properties](#)

[Overview](#)

[Properties](#)

[Findings](#)

[A01: A blocked address can unblock itself](#)

[Scenario](#)

[Recommendation](#)

[Status](#)

[Informative findings](#)

[B01:](#)

[Scenario](#)

[Recommendation](#)

[Properties Checking](#)

[Path analysis:](#)

Summary

[Runtime Verification, Inc.](#) has audited the smart contract source code of Gigastar Channel Revenue Distribution. The review was conducted from 2023-01-30 to 2023-02-17.

The Channel Revenue Distribution (CRD) contract aims to securitize revenue payments from a media platform to investors. Investors purchase Channel Revenue Tokens (CRT), representing a revenue percentage. The main goal of CRD is to provide a mechanism to distribute stablecoins periodically to investors according to the revenue percent of the CRT they own. The KYC (Know Your Customer) Contract is a ledger of registered accounts. When an investor registers a wallet, he is issued a KYC token. Only investors that own a KYC token can claim the revenue paid by the media platform.

The issues which have been identified can be found in section [Findings](#). Additional suggestions can be found in section [Informative findings](#).

Scope

The audited smart contracts are:

- `ICRD.sol`
- `CRD.sol`
- `IKYC.sol`
- `KYC.sol`
- `utils/IAccess.sol`
- `utils/IDiv.sol`
- `utils/IRoyalty.sol`
- `utils/ISecurity.sol`
- `utils/ISecurity721.sol`
- `utils/ITier.sol`
- `utils/IToken.sol`
- `utils/Access.sol`
- `utils/Checks.sol`
- `utils/Div.sol`
- `utils/Royalty.sol`
- `utils/String.sol`
- `utils/Tier.sol`
- `utils/Token.sol`

The audit has focused on the above contracts, and has assumed correctness of the libraries and external contracts they use. The libraries are widely used and assumed to be secure and functionally correct.

The review focused mainly on the GigaStarIo-public/polygon-crd private code repository. The code was frozen for review at commit [201569c8338054bac56ecf517424e0e006b08f77](#).

Assumptions

The audit is based on the following assumptions and trust model.

1. All addresses assigned a role need to be trusted for as long as they hold that role. These roles include: owner and manager.
2. The contracts are upgradeable. Thus, the admin with power to upgrade the contract, must be trusted, as he can significantly change the protocol's behavior.

These assumptions assume honesty and competence. However, we will rely less on competence and point out, wherever possible, how the contracts could better ensure that unintended mistakes cannot happen.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in [Disclaimer](#), we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to [known security issues and attack vectors](#). Finally, we met weekly with the GigaStar team, providing feedback and suggesting development practices and design improvements.

This report describes the **intended** behavior and invariants of the contracts under review. Then it outlines issues we have found, both in the intended behavior and how the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.



Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Channel Revenue Distribution: Contract Description and Properties

This section describes the Channel Revenue Distribution project at a high-level and which invariants we expect it to always respect at the end of a contract interaction.

Overview

The Channel Revenue Distribution (CRD) project's main goal is to tokenize revenue share from a media platform and provide a mechanism to distribute stablecoin monthly to investors. The CRD contract code is meant to facilitate a purchase agreement between an issuer and investors.

The issuer owns (or is responsible for) some media platform (a youtube channel, for instance) and agrees to share its revenue (up to some percentage) with investors. Investors will purchase tokens via Gigastar's website, where each token will have a tier (e.g. Gold, Platinum, Diamond). Each tier represents a different revenue percentage per token (e.g. 0.001%, 0.004%, 0.012%). If someone purchases 2 Gold and 1 Platinum, they would expect $(2 \times 0.001\% + 1 \times 0.004\%) = 0.006\%$ of the channel revenue in each distribution using a stablecoin (e.g. USDC).

Notice that when an investor purchases a CRT, they become the token's owner; however, owning the token does not entitle them to ownership in the underlying channel. The owner of a CRT is entitled to channel revenue paid by the media platform. The details of ownership and revenue distribution are given in the terms of the investment.

The purchase process (aka "the drop") is via a traditional web2 process, and the blockchain process begins after the drop closes. Scripts will construct a CRD based on parameters from the drop (e.g. channel info for metadata, tier info) and then create tokens for each investor. Every month when the channel pays revenue share - also called divs - an escrow agent will translate the USD to USDC and transfer it to the holding account, which is the source for a distribution to investors.

To be able to purchase CRT tokens, the investor has to go through a "know your customer" check. At this point the investor is issued a KYC token. The KYC contract is a ledger of registered accounts and is responsible for managing KYC tokens.

In order to receive the revenue share associated with their CRTs, the investor has to register a wallet. The wallet will be associated with the respective KYC token for the investor. Each KYC token has a `userId` that can be associated with off-chain data. A date and level (representing the level of KYC checks) are on-chain. A `userId` can also associate with multiple wallets, allowing

the CRD contract to distinguish between a trade (i.e. a transfer between different users) versus a same-user transfer (e.g. hot to cold storage).

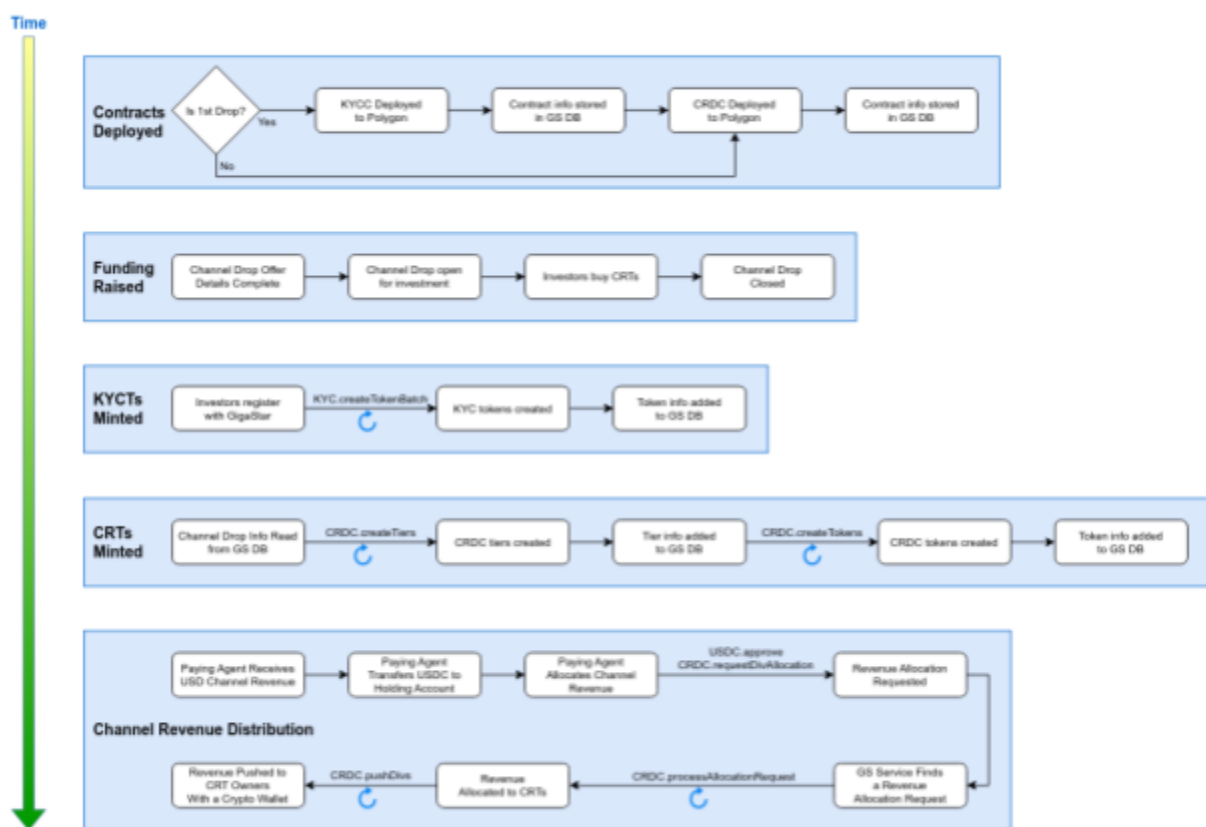


Image 1 - High-level workflow. Source: Gigastar

Suppose an investor does not have a registered account/wallet. In that case, the holding account will store its USDC, and Gigastar will store the ownership info in its database until a wallet is registered. At this point, the owner can claim all the divs that were accumulated before.

Both CRT and KYC tokens are NFTs, implemented through [OpenZeppelin ERC721Upgradeable](#). An investor can trade CRT tokens but restricted to some circumstances:

1. Transfer to another registered wallet - the new wallet must be associated with a KYC token;
2. Transfer to another user - the new owner's wallet must be associated with a KYC token. Can only be done after the holding period - usually a year;
3. Buy back - the token is purchased again by the issuer. Corresponds to a transfer to the holding account. The token can be later destroyed or purchased again. Can only be done after a holding period - usually a year;

The transfers between different users can only be performed through the Market contract - outside of the scope of the audit. When such a trade is performed, a percentage of the selling price goes to the royalty receivers - usually the issuer and Gigastar (but can be set to others). The transfer both between wallets of the same user and the buybacks are free from any fees.

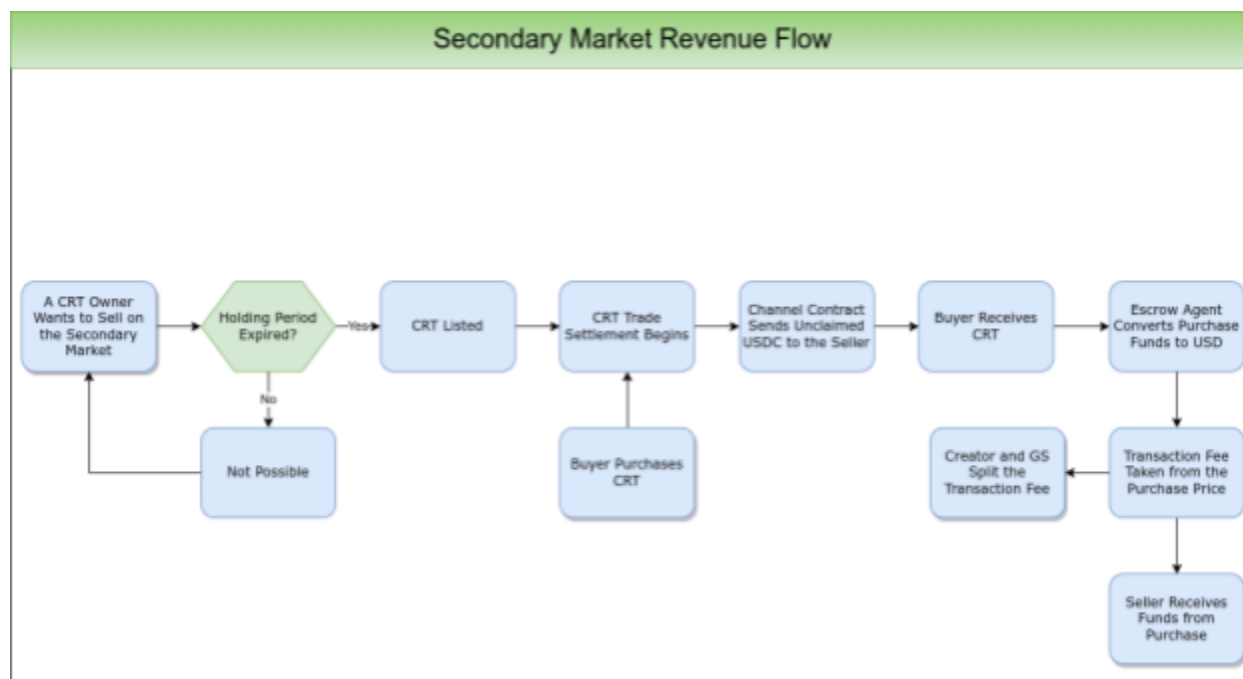


Image 2 - Source: Gigastar

Roles

❖ **Gigastar** - usually a royalty receiver. Responsible for the contracts management. There are two privileged roles owned by Gigastar: owner and manager. The separation between these roles was not following a specific criteria at the beginning of the audit:

- Owner - responsible for setting important parameters in the contracts, such as: royalty receivers, KYC contract used by CRDC and the holding period;
- Manager - responsible for divs distribution, pausing/unpausing functionality and tokens management;

To automate the process of deploying the contracts and minting the tokens, Gigastar intended to have both, owner's and manager's, private keys stored in the server. See vulnerability [AO1](#), which addresses this problem, proposing a new separation between owner and manager.

❖ **Issuer/Creator** - the owner or the responsible of the media platform/ channel;

- ❖ **Investor** - owner of CRT tokens and receiver of divs distribution; the only method accessible that changes state is transferring CRT tokens to another wallet associated with the same investor.
- ❖ **Holding account** - account from where the stablecoin (usually USDC) is transferred, and that will be the source for the divs distribution. It is not clear if the owner of this account is Gigastar or the issuer yet. Both entities should be trustworthy as they have a signed agreement they must follow against the penalty of being prosecuted in US court.

Properties

Several important properties should always be satisfied by the contract. Here will only be listed properties related to the contracts under the audit. Properties related to the use of external libraries are assumed to hold.

In the following, we list several properties that the contract should satisfy. These are not the only ones, but they are fundamental for the correctness of the protocol and as such, deserve special attention.

Channel Revenue Distribution

1. The privileged addresses - owner and manager - are valid addresses, i.e. are different than `address(0)` and, ideally, different from each other
2. The only public methods that change state that non-privileged addresses can call are `transferFrom` or `safeTransferFrom` a token to another wallet owned by the same user, and `approve` and `setApprovalForAll`. All the other functions that change state are only accessible to the owner or manager (or both).
3. If the contract is paused, all the functionalities, except the ones only accessible to the owner, should not be available
4. If the contract is unpaused, all the functionalities should be available.
5. Only the owner should be able to upgrade the contract
6. The `_holding` account is always different from `address(0)`
7. $\forall_{\text{token}} \text{token.owner} \neq \text{address}(0)$
8. $\forall_{\text{token}} \text{_kyc.balanceOf(token.owner)} \neq 0$
9. `_brokenTokenId == 0`, except if `fixTokenOwnership` is being executed

10. If `fixTokenOwnership(newOwner, tokenId)` is being executed, then
`_brokenTokenId == tokenId`, for some `newOwner` and `tokenId`, i.e.:
 - `_brokenTokenId == tokenId ↔ fixTokenOwnership(newOwner, tokenId)`
11. If `tokenId ∈ _buybacks` then `exists(tokenId)`, which means that all tokens in
`_buybacks` were minted before and were not burned yet
12. If `tokenId ∈ _buybacks` then `getToken(tokenId).owner == _holding`.
 Equivalent to:
 - If `getToken(tokenId).owner != _holding` then `tokenId ∉ _buybacks`
13. If `tokenId ∈ _buybacks` then `getToken(tokenId).holdPeriodEnd` has
 passed, which means it is only possible to buy back after holding period.
14. Only tokens that are in `_buybacks` can be burned, which implies that a token can
 only be burned if the `getToken(tokenId).holdPeriodEnd` has passed
15. It is only possible to trade a given token if the `getToken(token).holdPeriodEnd`
 has passed
16. `exists(tokenId) ↔ getToken(tokenId).tierId != 0`
17. `exists(tokenId) ↔ _tokens[tokenId] != 0`
18. `exists(tokenId) ↔ tokenId ∈ getToken(tokenId).tierId.tokenIds`
19. $\forall_{tokenId} \text{ tokenId} \in \text{getToken(tokenId).tierId.tokenIds}$
20. $\forall_{tokenId, tier \neq \text{getToken(tokenId).tierId}} \text{ tokenId} \notin \text{tier.tokenIds}$
21.
$$_issuedPercent = \sum_{tier=0}^{getTierCount} \text{tier.tokenIds.count} * \text{tier.tokenRevPercent}$$
22. `_issuedPercent ≤ _issuedPercentMax`
23. $\forall_{tokenId} \text{ getToken(tokenId).owner} == \text{ownerOf(tokenId)}$
24. $\forall_{CRDToken} \exists_{KYCToken} \text{ kyc.tokens[KYCToken].owner} == \text{getToken(CRDToken).owner}$
 $\&\& \text{ kyc.tokenIds[getToken(CRDToken).owner]} == \text{KYCToken}$

Div.sol

25. `_div` is different from `address(0)`

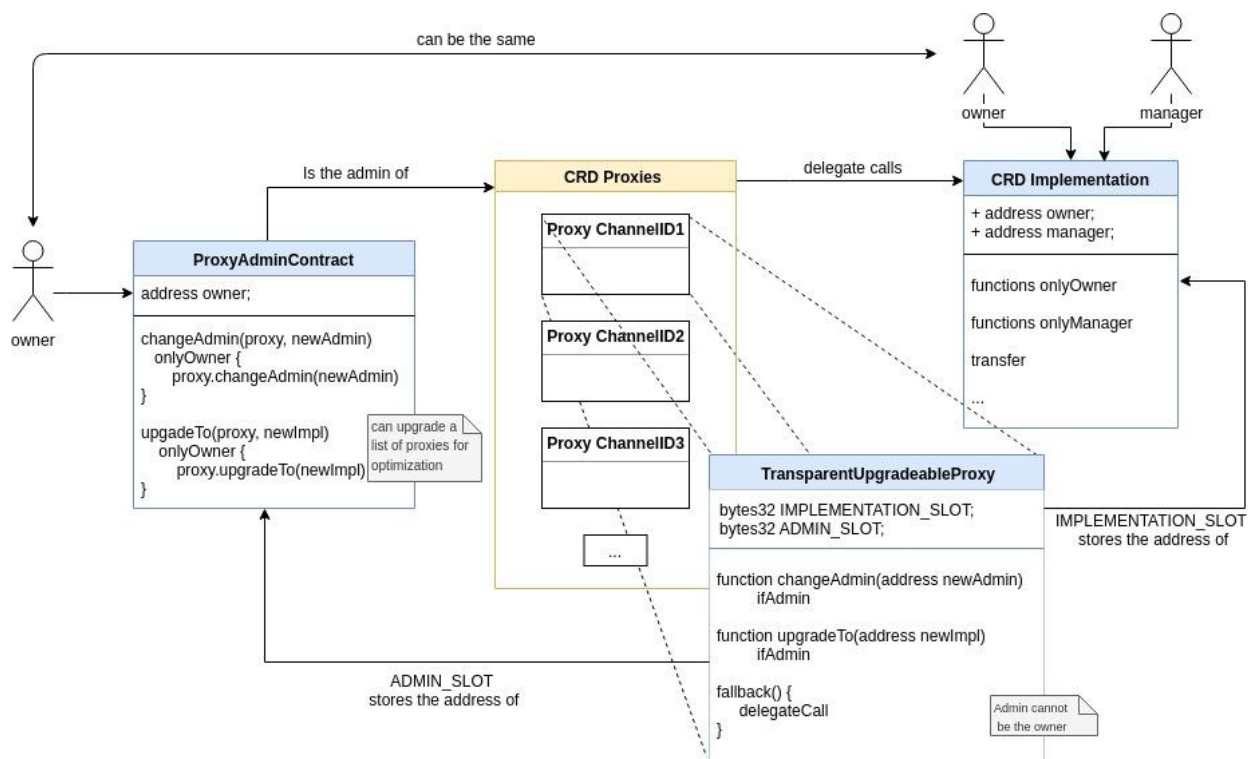
26. If there is no request in progress then `_divAllocReq.nextTierIndex == 0` && `_divAllocReq.nextTokenIndex == 0`
27.
$$\text{divAmount} \geq \sum_{\text{tier} = 0}^{\text{getTierCount}} \text{tier.tokenIds.count} * \text{divPerToken}$$
28. `_divAllocReq.divAmountIn == _divAllocReq.sumAllocated + _divSlip`

Upgradeability

It is intended for the contracts to be upgradeable. However it was not clear which upgradeability pattern would be used. Addressing this was one concern of this audit as well.

The CRD contract has almost the contract size limit. Therefore, the [UUPS Upgradeable](#) pattern cannot be used. This pattern implies the implementation includes, alongside the contract's logic, all the code necessary to update the implementation's address stored at a specific slot in the proxy's storage space. However the space left in the CRD contract is insufficient to include the upgradeability logic.

For each CRD channel, it should be created a new proxy. Each proxy handles the underlying channel's state and delegates the calls to the implementation address. Since the UUPS pattern is not an option, we think the [Transparent](#) pattern should be used. Notice that this pattern has the `admin` role, and only the `admin` can upgrade the contract, i.e. change the implementation address stored in `IMPLEMENTATION_SLOT`. However, the `admin` must not be any of the addresses intended to call functions on the contract, i.e. the `admin` cannot be the owner or the manager. Otherwise, all the functionality restricted to owner or manager will be lost. The `admin` account can only be used for admin actions like upgrading the proxy or changing the `admin`, so it's best if it's a dedicated account that is not used for anything else. OpenZeppelin [recommends](#) the dedicated account to be an instance of the [ProxyAdmin](#) contract. Following is an upgradeability scheme suggestion:




Each channel is a `TransparentUpgradeableProxy` which admin is a [ProxyAdmin](#) contract. The `ProxyAdmin` contract is Ownable, and the owner is the only one responsible for upgrading the contract.

Notice that with this approach, it is possible the entity responsible for upgrading the contract - the owner of `ProxyAdmin` - and the owner of the CRD contract to be the same.

Gigastar planned to use the [OppenZeppelin Hardhat Upgrades](#) plugin in their deployment. By default, it deploys Transparent proxies and creates a single `ProxyAdmin` for all the proxies, similar to the diagram above. According to their [documentation](#), the owner of the `ProxyAdmin` contract is, by default, the externally owned account used during deployment - which means that the manager is the entity with the power to upgrade the contract (violates P5 - see finding [A04](#)).

To upgrade the contract, each proxy has to be updated individually. To optimize that, it is possible to use the [Beacon Proxy](#) pattern. This contract implements a proxy that gets the implementation address for each call from an [UpgradeableBeacon](#) contract. To upgrade all the proxies, one must update the implementation address in the `UpgradeableBeacon` contract. This approach is more convenient to upgrade all the proxies in just one step. However, each call to the implementation will be slightly more expensive since it needs to read the implementation address from an external contract - we encourage testing and choosing the most suitable option.



In conclusion, we advise reading more about proxies and deepening your knowledge in upgradeability patterns. After setting your deployment process, it is strongly recommended to audit the deployment scripts since some vulnerabilities can happen at this stage.

Findings

AO1: owner and manager private keys stored in server

[Severity: High | Difficulty: Medium | Category: Security]

In order to automate transactions, Gigastar planned to store owner and the manager private keys in the server. This would make it easier for scripts to read the PKs and make the transactions. However, this approach makes the contract vulnerable to some attack on the server. If some untrusted user can access the server, he could take over the contract. There is no recovery mechanism for this scenario. Gigastar realized problems with this approach but it was meant to be a stop-gap to focus the limited time on features and resolve this issue pre-production.

Scenario

If some untrusted user gets access to the server, he can:

- Transfer tokens to some account of his choice leading to loss of funds for all token owners;
- Mint new tokens;
- Arbitrarily burn tokens;
- Set the owner and the manager to addresses of his choice - Gigastar loses total control over the contracts;
- Upgrade the contract.

Recommendation

We recommend not storing the PK of the owner in the server but in a safer place, protected with multisig. This way, if the manager gets compromised, the owner can still recover by setting a new manager - assuming property P1 holds (after addressing AO2).

We discussed this issue with Gigastar, who was aware of the issue. However, it is important for them to automate the process through scripts that read drop information from Gigastar databases and create the necessary transactions. After some back-and-forth discussion with Gigastar, with suggestions from both sides, the following solution seemed to best accommodate the security concerns and the desired efficiency:

1. Change the separation between owner and manager and which functions they can call:
 - **Owner** - only the owner can perform critical operations such as: upgrade the contracts, pause and unpause, mint or burn tokens (CRT and KYCT), transfer

tokens (CRT and KYCT), fix CRT token ownership, update KYC tokens and set critical parameters (kyc address in CRD, holding period, URI base and the royalty receivers);

- **Manager:** perform less critical operations such as revenue distribution. The `manager` would be responsible for: `requestDivAlloc`, `processAllocRequest`, and `pushDivs` functions. An untrusted manager can still call `requestDivAlloc` with a higher amount than what it is supposed to. However, the amount is bounded by the balance of the `_holding` account and the allowance the `_holding` account has given to the contract. According to the business model, the holding account should only have the balance to distribute to investors.
2. Store the `manager` PK in the server to automate the transactions related to divs distribution. Store the owner PK in a safer place, protected with multisig.
 3. Allow the `manager` to be the owner during the setup process, i.e. after the drop, when it is necessary to mint KYC tokens for investors that don't have one yet, `createTiers`, and mint CRT tokens for the investors. This process can be automated by the scripts that read drop information from Gigastar databases and create the necessary transactions.
 4. After the setup process, the `manager` transfers ownership to the owner. From then on, the `manager` can only perform divs distribution, and only the owner can perform critical operations on the contract.

With this approach, even if the manager gets compromised, that does not imply loss of funds for the investors. Besides, the owner can pause the contract, set a new `manager`, and recover from the situation.

If the owner gets compromised, all the contract gets compromised. However, notice that with this approach, the chances of that happening are significantly smaller.

Status

Addressed in commits: [86f8164](#) , [46b53df](#) and [721538b](#).

AO2: owner and manager can be the same address

[Severity: High | Difficulty: Medium | Category: Security / Input validation]

In `Access.sol`, the function `setOwner` does not check the new owner account is different from the actual manager.

```
function setOwner(address account) external override onlyOwner {
    address from = _owner;
    if(account == from) return;
    _checkAddress(Tag.account, account);
    _owner = account;
    grantRole(ROLE_MANAGER, account); // emits RoleGranted
    revokeRole(ROLE_MANAGER, from);
}
```

Scenario

1. The current owner calls `setOwner(manager)`
2. The new owner is the manager, whose PK is stored in the server
3. The server gets compromised
4. The attacker has complete control over the contract

Recommendation

We think it is essential that the invariant `owner != manager` holds. Since the PK for the manager is stored in the server, if the manager is also the owner, there is no recovery process if the manager gets compromised. It is crucial to keep the PK of the owner different and safely separate; if the manager gets compromised, the owner can still recover by setting a new manager.

Status

Addressed in commit [86f8164](#). Now the function `setOwner` checks that the `newOwner` does not have the `ROLE_MANAGER`, which implies that the `newOwner` is different from the manager.

```
if(oldOwn == newOwn || hasRole(ROLE_MANAGER, newOwn)) return;
```

We still recommended that the function should `revert` instead of `return` - which was addressed in commit [46b53df](#).

AO3: Missing check in Div.sol

[Severity: Low | Difficulty: - | Category: Usability / Input validation]

In `Div.sol`, the function `__Div_init` does not check the `div` argument differs from `address(0)`. Since there is no function to modify the `_div` variable later, all the functionality to distribute `divs` would be unavailable, and the contract would have to be deployed again.

Recommendation

Add a check that the `div` argument differs from `address(0)` before assigning it to `_div`.

Status

Addressed in commit [46b53df](#).

AO4: manager is responsible for upgrading the contract

[Severity: High | Difficulty: Medium | Category: Security]

The deployment plan was to use the OpenZeppelin Hardhat Upgrades plugin. This plugin creates a ProxyAdmin contract which is the admin of all proxies. According to their [documentation](#), the owner of the ProxyAdmin contract is, by default, the externally owned account used during deployment - which will be the manager. This violates property P5, since only the owner should have the power to upgrade the contract. Besides, since the manager's private key is being stored in the server, the attacker has complete control over the contract if the server gets compromised because he can upgrade it as he wants.

Recommendation

In the end of the setup process (see [AO1](#) status), along with transferring ownership of the CRD contract from manager to owner, transfer the ownership of the ProxyAdmin contract from manager to owner calling the transferProxyAdminOwnership function.

AO5: owner can setOwner to invalid address

[Severity: - | Difficulty: - | Category: Input validation]

The owner can call `setOwner` giving as argument an invalid address which leaves the contract without an owner, thus removing any functionality that is available only to the owner.

Scenario

The owner calls the `setOwner` function giving as argument an invalid address. This will make it not possible to call the `onlyOwner` functions anymore. After addressing [A01](#), this would mean that a significant part of the contract's functionality would become unavailable.

Recommendation

This is not a security concern under the assumption that the owner is honest and competent. Nevertheless, we want to make it clear that this undesired behavior is possible. To make this behavior impossible, we recommend implementing a two-step transfer ownership (similar to [this](#)).

Informative findings

Bo1: balanceOf function does not revert as expected

[Severity: - | Difficulty: Low | Category: Protocol Invariants]

In KYC.sol, the function balanceOf is expected to revert for an unknown owner, but that is not true. The super.balanceOf function returns 0 if the tokenOwner does not exist.

```
function balanceOf(address tokenOwner)
    public view override(ERC721Upgradeable, IERC721Upgradeable) returns(uint256)
    {
        _requireNotPaused();
        return tokenOwner == address(0) ? 1 : super.balanceOf(tokenOwner); // super
call reverts if not found
    }
```

This function is being used in _beforeTokenTransfer function in CRD.sol to ensure that transfers of CRT tokens only happen between owners that also have a KYC token.

```
_kyc.balanceOf(to); // reverts for an unknown account
if(!isMint) {
    _kyc.balanceOf(from); // reverts for an unknown account
```

However, this check is not necessary because if some user does not have KYC token, the call to the function kyc.isSameUser in _beforeTokenTransfer would revert.

Recommendation

Delete the calls to balanceOf function in _beforeTokenTransfer. Update the comment in balanceOf function in KYC.sol.

Status

Addressed in commit [46b53df](#).

Bo2: fixTokenOwnership missing check

[Severity: - | Difficulty: - | Category: Input validation / Documentation]

In `CRD.sol`, the `fixTokenOwnership` function does not check that the token is transferred to the same user.

Also, the KYC token with the bad wallet is not being deleted. Therefore, the user has no penalty in claiming that he lost his wallet and transfers the tokens to another user bypassing the secondary market fees.

Scenario

1. Alice registers Gigastar with `accountA`
2. Bob registers Gigastar with `accountB`
3. Alice purchases CRT tokens
4. Alice claims that she lost her wallet and asks Gigastar to transfer tokens to her new wallet: `accountB`

Recommendation

Either document properly why the check is not being done or add a check that ensures that the token is being transferred to the same user: `kyc.areSameUser(newOwner, token.Owner)`.

Status

After discussing this issue with Gigastar, they clarified that `fixTokenOwnership` could also be used in fraudulent purchases (the payment for the token was not finalized). In this case the token will be transferred to the `_holding` account. Therefore, the check cannot be added to the function; otherwise, it would be impossible to use it in case of fraudulent purchases. Besides, GigaStar clarified that in case some user claims having lost his wallet, there will be an off-chain check that the new wallet he wants his tokens transferred to also belongs to him.

Regarding the KYC token management, Gigastar justifies that it will occur off-chain since the bad wallet could affect multiple CRTs. Each CRT requires a KYC token for a transfer during `fixTokenOwnership`. The pseudo-code sequence would be:

```
userCRTs(userA.badAddress).forEach(fixTokenOwnership);  
kyc.deleteToken(userA.badAddress)
```

Bo3: Unnecessary check in `_beforeTokenTransfer`

[Severity: - | Difficulty: - | Category: Optimization]

In `CRD.sol`, the function `_beforeTokenTransfer` has the following if statement:

```
if(to != _holding) // Not an unclaim
  _pushDiv(tokenId, to); // isBurn would fail to transfer
```

However it is not necessary to check that `to != _holding` to call the `_pushDiv` function, because in case `to == holding` the function returns before changing the state.

Recommendation

Remove the unnecessary check.

Status

Addressed in commit [a75407c](#).

Bo4: remove unnecessary condition in setUriBase

[Severity: - | Difficulty: - | Category: Optimization]

In `CRD.sol`, the function `setUriBase` has the following unnecessary `if` statement:

```
else if(bytes(_uriBase).length == 0)
    return;
```

Recommendation

Remove the `if` statement.

Status

Addressed in commit [8a1cfca](#).

Bo5: Optimization in destroyBuyBacks

[Severity: - | Difficulty: - | Category: Optimization]

In CRD.sol, the function destroyBuyBacks has the following unchecked loop:

```
unchecked {
    uint256 end = pageSize >= tokenCount ? 0 : tokenCount - pageSize;
    for(uint256 i = tokenCount - 1; i >= end; --i) {
        _destroyToken(tokenIds[i] = EnumerableSetUpgradeable.at(_buyBacks, i));
        if(i == 0) break;
    }
}
```

Since the end can be 0 and the loop is inside unchecked, it is necessary to check in each iteration `if(i==0) break;`, otherwise there could be an infinite loop.

Recommendation

The for loop can be replaced by the following while loop, with no need for the `if` statement inside the loop, thus saving gas:

```
unchecked {
    uint256 end = pageSize >= tokenCount ? 0 : tokenCount - pageSize;
    uint256 i = tokenCount;
    while(i > end) {
        i--;
        _destroyToken(tokenIds[i] = EnumerableSetUpgradeable.at(_buyBacks, i));
    }
}
```

Status

Addressed in commit [969f48b](#).

Bo6: Optimization in checkUriBase function

[Severity: - | Difficulty: - | Category: Optimization]

In `Checks.sol`, the function `checkUriBase` does not use the `uri` variable and instead re-casts the string value to bytes in the code.

```
function _checkUriBase(string memory value) pure {
    bytes memory uri = bytes(value);
    if(bytes(value).length == 0)
        revert EmptyString(Tag.uriBaseLength);
    if(bytes(value).length < 100 && uri[uri.length - 1] != '/')
        revert UriBase(value);
}
```

Recommendation

Use the `uri` variable instead of re-casting the string value to bytes.

Status

Addressed in commit [969f48b](#).

Bo7: Unnecessary assignment in fixTokenOwnership

[Severity: - | Difficulty: - | Category: Optimization]

In CRD.sol, the function fixTokenOwnership assigns the token.owner to the newOwner:

```
function fixTokenOwnership(address newOwner, uint256 tokenId) external override
onlyOwner nonReentrant {
    TokenInfo storage token = _getToken(tokenId);
    if(newOwner == token.owner) return;
    address prevOwner = token.owner;
    token.owner = newOwner;
```

However, this instruction is not necessary since when transferring the token to the newOwner through safeTransferFrom(prevOwner, newOwner, tokenId), the function beforeTokenTransfer will be called and will perform the assignment:

```
if(!isBurn)
    _getToken(tokenId).owner = to;
```

Notice that we can be sure that !burn is true, because if new == address(0) then the function safeTransferFrom would revert.

Recommendation

Remove the assignment token.owner = newOwner from fixTokenOwnership function.

Status

Addressed in commit [969f48b](#).

Bo8: Unnecessary assignment in pushUnclaimedTokens

[Severity: - | Difficulty: - | Category: Optimization]

In `CRD.sol`, the function `pushUnclaimedTokens` assigns the `token.owner` to the `newOwner`:

```
_getToken(tokenIds[i]).owner = newOwners[i];
```

Similarly to [Bo7](#), the assignment is unnecessary since the function `beforeTokenTransfer` will perform it.

Recommendation

Remove the assignment `_getToken(tokenIds[i]).owner = newOwners[i]` from the `pushUnclaimedTokens` function.

Status

Addressed in commit [969f48b](#).

Bog: Optimization in token transfers

[Severity: - | Difficulty: - | Category: Optimization]

In `Div.sol`, the function `_pushDivSub` checks that the transfer is authorized:

```
if(!_isTransferAuthorized()) revert UnauthorizedDivPush(msg.sender);
```

This function is called when:

- The `pushDivs` function is called in `CRD.sol` - we know that the transfer is authorized because `pushDivs` is only accessible to the `manager`;
- The `_pushDiv` function is called in `_beforeTokenTransfer`:
 - When `isTokenFix` is true. This means that the function `fixTokenOwnership` was called, and therefore, the transfer is authorized because `fixTokenOwnership` is only accessible to the `manager`;
 - When `!isSameUser` is true. This means it is either a claim, a trade, or a buyback. During a trade and a buyback (which is always also a trade), it is checked that the transfer is authorized:

```
_checkTransferAuthorized();
```

Recommendation

If we check that the transfer is authorized for all transfers between different users, i.e., move the check done at line 182 to line 178, then the check performed in `_pushDiv` line 176 would be unnecessary.

Status

Addressed in commit [46b53df](#).

B10: Reorder if statements in merge

[Severity: Low | Difficulty: - | Category: Optimization]

In String.sol, the merge function has the following if statement that checks the validity of the offset input:

```
if(offset % WORD != 0) revert BadOffset(offset);
```

Recommendation

The if statement can be moved to the top of the function.

Status

Addressed in commit [46b53df](#).

B11: Avoid using magic values

[Severity: Low | Difficulty: - | Category: Best Practice]

The function `_setUriLength` in `Tier.sol` has the following comparison with a concrete value:

```
if(uriLength < 32 || URI_MAX_BYTES < uriLength) revert BadURLength(uriLength);
```

Recommendation

Declare the value 32 as a constant.

Status

Addressed in commit [46b53df](#).

Properties Checking

- P1. The privileged addresses - owner and manager - are valid addresses, i.e. are different than `address(0)` and, ideally, different from each other
- True if [A02](#) is addressed;
- P2. The only public methods that change state that non-privileged addresses can call are `transferFrom` or `safeTransferFrom` a token to another wallet owned by the same user, and `approve` and `setApprovalForAll`. All the other functions that change state are only accessible to the owner or manager (or both).
- True, all the other methods that change state have the `onlyOwner` or `onlyManager` modifier
- P3. If the contract is paused, all the functionalities, except the ones only accessible to the owner, should not be available
- If [A01](#) and [A02](#) are addressed, then the functions to distribute divs - `requestDivAlloc`, `processAllocRequest` and `pushDivs` - are the only ones that are accessible to the manager. However these functions are still available if the contract is paused, which makes this property not hold.
- P4. If the contract is unpaused, all the functionalities should be available.
- True. The only functionality that requires that the contract is paused is the `_unpause`; however, unpausing an unpaused contract would leave it in the same state.
- P5. The owner and only the owner should be able to upgrade the contract
- True if [A04](#) is addressed.
- P6. The `_holding` account is always different from `address(0)`
- True. The only function responsible for changing the `_holding` address - `setHolding` - checks that the argument is different from `address(0)`.
- P7. $\forall_{\text{token}} \text{token.owner} \neq \text{address}(0)$
- True:
 - a. When tokens are created, a call to `_safeMint` reverts if `owner == address(0)`.
 - b. Every time there is an assignment to the `token.owner`, there is also a call to `safeTransferFrom` which ensures that the `to` \neq `address(0)`, i.e.

`newOwner != address(0)`. The only exception is the assignment in function `_beforeTokenTransfer`; however this assignment is inside an `if` statement that checks `!isBurn` is true, i.e. `to != address(0)`.

P8. $\forall_{token} \text{ _kyc.balanceOf(token.owner) } \neq 0$

- True if [AO4](#) is addressed.

P9. `_brokenTokenId == 0`, except if `fixTokenOwnership` is being executed

- True. `_brokenTokenId == 0` in the initial state, and only 2 assignments change its value:
 - In the `fixTokenOwnership` function, assigning it to a value different than 0
 - In the `_beforeTokenTransfer` function, assigning it to value 0. The `_beforeTokenTransfer` function is (indirectly) called at the end of the function `fixTokenOwnership`, which means that outside of the execution of this function the value of `_brokenTokenId` is always 0.

P10. `_brokenTokenId == tokenId ↔ fixTokenOwnership(newOwner,tokenId)`

- True. See explanation for P9.

P11. $\forall_{tokenId}$ If `tokenId ∈ _buyBacks` then `exists(tokenId)`, i.e. all tokens in `_buyBacks` were minted before and were not burned yet

- True. `_buyBacks` is empty in the initial state. A token is only added to the `_buyBacks` in the `_beforeTokenTransfer` function if the flag `isMint` is false and the flag `isBuyBack` is true, which means that:
 - `from != address(0)` - not a mint
 - `to == _holding` - not a burn

Which means that the function `_beforeTokenTransfer` was called during a transfer. The `_transfer` function ensures that:

- `ownerOf(tokenId) == from`

From a. and c. we get: `ownerOf(tokenId) != address(0)` which is equivalent to `exists(tokenId)`

P12. $\forall_{tokenId}$ If `tokenId ∈ _buyBacks` then `getToken(tokenId).owner == _holding`

- True. `_buyBacks` is empty in the initial state. A token is only added to the `_buyBacks` in the `_beforeTokenTransfer` function if the flag `isBuyBack` is true, which means that `to == _holding`. We know from P6 that `_holding != address(0)`, which means that the flag `isBurn` is false; therefore, the function

`_beforeTokenTransfer` will also make the assignment:
`getToken(tokenId).owner = _holding`

P13. $\forall_{tokenId}$ If `tokenId \in _buyBacks`, then `getToken(tokenId).holdPeriodEnd` has passed, which means it is only possible to buy back after holding period.

- True. `_buyBacks` is empty in the initial state. A token is only added to the `_buyBacks` in the `_beforeTokenTransfer` function if the flag `isBuyBack` is true. Notice that if `isBuyBack` is true, then `isTrade` is also true, which means that `_checkHoldPeriod(tokenId)` will be executed, guaranteeing that a token is only added to the buybacks after holding period.

P14. Only tokens `tokenId \in _buyBacks` can be burned, which implies that a token can only be burned if the `getToken(tokenId).holdPeriodEnd` has passed

- True. The function `_burn` is only called inside the `_destroyToken` function, which, in its turn, is only called when iterating over the `_buyBacks` in the `destroyBuyBacks` function. From P13, tokens can only be burned after holding period.

P15. It is only possible to trade a given token after `getToken(token).holdPeriodEnd`

- True. A trade is a transfer between 2 users. The function `_beforeTokenTransfer` ensures through `_checkHoldPeriod(tokenId)` that a transfer can only happen after the holding period.

P16. `exists(tokenId) \leftrightarrow getToken(tokenId).tierId != 0`

- True. Tokens are only created in `_createTokens` function. When a token is created - `_createToken(owner, tierId, holdPeriodEnd)`, we know that `tierId != 0`, otherwise, `_getTier(tierId)` would revert. After the token is created, the token is minted - `_safeMint(owner, _tokenIdSeed)`, and that's the only place where tokens are minted. Tokens are only burned in the `_destroyToken` function. When a token is burned - `_burn(tokenId)`, it also gets deleted - `_deleteToken(tokenId)`. Besides `getToken(tokenId).tierId` is not changed anywhere else in the code. Therefore the property holds.

P17. `exists(tokenId) \leftrightarrow _tokens[tokenId] != 0`

- True. Tokens are only added to `_tokens` in the `_createToken` function, and only deleted from `_tokens` in the `_deleteToken` function. Therefore, if P16 holds, then P17 also holds.

P18. `exists(tokenId) \leftrightarrow tokenId \in getToken(tokenId).tierId.tokenIds`

- True. Tokens are only minted in the `_createTokens` function. Right before a token is minted, it is added to `tokenId.tokensIds`. A token is only burned in the `_destroyToken` function, where it is also removed from `getToken(tokenId).tokenId.tokensIds`. Therefore, the property holds.

P19. $\forall_{tokenId} \text{tokenId} \in \text{getToken(tokenId).tokenId.tokensIds}$

- True. When a token is created, it is also added to `_getTier(tierId).tokensIds`.

P20. $\forall_{tokenId, tier \neq \text{getToken(tokenId).tokenId}} \text{tokenId} \notin \text{tier.tokensIds}$

- True. A token is only added to the `tokensIds` of its respective tier - `getToken(tokenId).tokenId`, when it is created. Anywhere else in the code, the `tokenId` is added to any other tier. Besides, `getToken(tokenId).tokenId` is never modified until the token gets deleted.

P21. $_issuedPercent = \sum_{tier = 0}^{getTierCount} \text{tier.tokensIds.count} * \text{tier.tokenRevPercent}$

- True. `_issuedPercent` is updated:
 - In function `_createTokens` with:
`_issuedPercent += tier.tokenRevPercent * owners.length;`
 Notice that this function will create a new token for each owner and add the `tokenId` to the `tier.tokensIds`.
 - In function `_destroyToken` with:
`_issuedPercent -= tier.tokenRevPercent;`
 But, right before updating `_issuedPercent`, the token is removed from `tier.tokensIds`.

P22. $_issuedPercent \leq _issuedPercentMax$

- True. The function `_createTokens`, the only function where `_issuedPercent` is increased, ensures after increasing `_issuedPercent` that the new value `_issuedPercent ≤ _issuedPercentMax`.

P23. $\forall_{tokenId} \text{getToken(tokenId).owner} == \text{ownerOf(tokenId)}$

- True. Tokens are created and minted at the same time, so the property holds after token creation. Tokens are deleted and burned at the same time, so the property holds when the tokens are destroyed. When tokens are transferred to a `newOwner`, the `_transfer` calls `_beforeTokenTransfer`, which assigns `getToken(tokenId).owner` to the `newOwner`. The `_transfer` function then assigns `_owners[tokenId]` to the `newOwner`. Therefore, the property holds after

token transfers. If [Bo7](#) and [Bo8](#) are addressed, there aren't any other assignments of `getToken(tokenId).owner`.

P24. $\forall_{\text{CRDToken}} \exists_{\text{KYCToken}} \text{kyc.tokens}[\text{KYCToken}].\text{owner} == \text{getToken}(\text{CRDToken}).\text{owner}$
 $\&\& \text{kyc.tokenIds}[\text{getToken}(\text{CRDToken}).\text{owner}] == \text{KYCToken}$

- True if [Ao4](#) is addressed.

P25. `_div` is different from `address(0)`

- True if [Ao3](#) is addressed.

P26. If no request is in progress, then `_divAllocReq.nextTierIndex == 0` && `_divAllocReq.nextTokenIndex == 0`

- True. In the initial state `_divAllocReq.nextTierIndex == 0` and `_divAllocReq.nextTokenIndex == 0`. The function `_processAllocRequest` assigns `nextTokenIndex` to 0 when it finishes processing all the tokens in some tier and assigns `nextTierIndex` to 0 after processing all the tiers. Notice that the function `_processAllocRequest` iterates over `tierCount * MAX_DIV_AMOUNTS`, with `MAX_DIV_AMOUNTS == 2` (the number of requests in progress). This means that `_processAllocRequest` will finish processing the distribution of `_divAllocReq.divAmountIn`, the current request in progress. After it finishes, it will process `_divAllocReq.nextDivAmount`, the next requests to process. Therefore, after finishing all the tiers, no request is in progress.

P27. $\text{divAmount} \geq \sum_{\text{tier} = 0}^{\text{getTierCount}} \text{tier.tokenIds.count} * \text{divPerToken}$

- True.

a. $\text{divPerToken} = \frac{\text{divAmount} * \text{tier.tokenRevPercent}}{\text{getIssuedPercent}()}$


b. From 21, we have:

$$\text{issuedPercent} = \sum_{\text{tier} = 0}^{\text{getTierCount}} \text{tier.tokenIds.count} * \text{tier.tokenRevPercent}$$

c. Therefore: $\text{divPerToken} = \frac{\text{divAmount} * \text{tier.tokenRevPercent}}{\sum_{\text{tier} = 0}^{\text{getTierCount}} \text{tier.tokenIds.count} * \text{tier.tokenRevPercent}}$

d. Therefore:

$$\text{divAmount} \geq \sum_{\text{tier} = 0}^{\text{getTierCount}} \text{tier.tokenIds.count} * \frac{\text{divAmount} * \text{tier.tokenRevPercent}}{\sum_{\text{tier} = 0}^{\text{getTierCount}} \text{tier.tokenIds.count} * \text{tier.tokenRevPercent}}$$


$$\equiv \mathit{divAmount} \geq \mathit{divAmount}$$

P28. `_divAllocReq.divAmountIn == _divAllocReq.sumAllocated + _divSlip`

- True.

Analysis of _beforeTokenTransferFunction

The function `_beforeTokenTransfer` is a critical piece in the CRD contract, since it will determine under what circumstances a transfer can occur. The function is large and has complicated logic: it has a set of boolean flags and a flow of if statements according to those flags. Given that, it seemed to us that the function could be error prone and required a deeper analysis.

The first step was to create a truth table with all the possible values for all the flags. The colored lines are the satisfiable ones, whereas the gray lines cannot happen. For instance, it is not possible that `isMint` and `isBurn` are true simultaneously. Secondly, we elaborated a control flow graph depicting all the possible paths according to the values of the boolean flags. And finally, we performed a path analysis exposing for each path what block code is being executed. This analysis allowed us to find some issues and improvements that could be done, already presented in the section [Findings](#) and [Informative Findings](#).

Truth Table

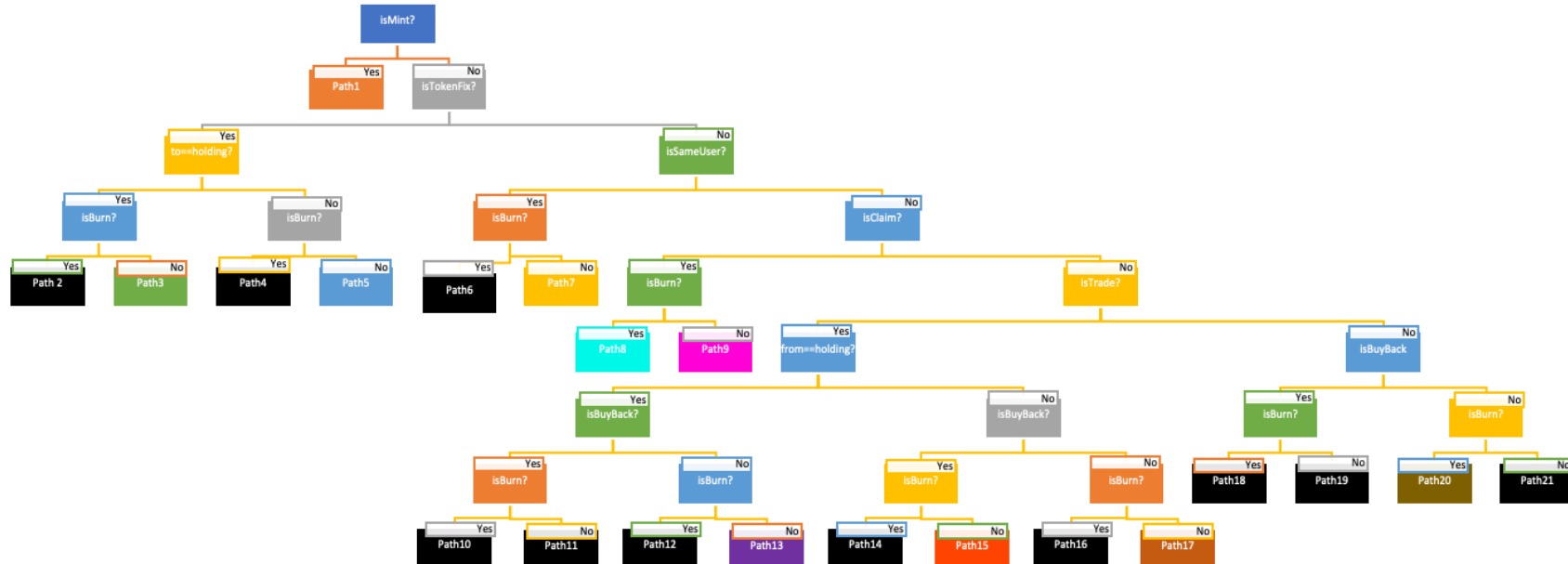
isMint	isBurn	isClaim	isSameUser	isTokenFix	isTrade	isBuyBack
1	0	0 && 1 = 0	0	0	0	1 && 0 = 0 Path1 to == holding
1	0	0 && 1 = 0	0	0	0	0 && 0 = 0 Path1
0	0	0 && 1 = 0	1	1	0	1 && 0 = 0 Path3 to == holding
0	0	0 && 1 = 0	0	1	0	1 && 0 = 0 Path3 to == holding
0	0	1	1	1	0	0 && 0 = 0 Path5 from == holding
0	0	1	0	1	0	0 && 0 = 0 Path5 from == holding
0	0	0 && 1 = 0	1	1	0	0 && 0 = 0 Path5
0	0	0 && 1 = 0	0	1	0	0 && 0 = 0 Path5
0	0	1	1	0	0	0 && 1 = 0 Path7 from == holding
0	0	0 && 1 = 0	1	0	0	1 && 1 = 1 Path7 to == holding
0	0	0 && 1 = 0	1	0	0	0 && 1 = 0 Path 7
0	0	1 && 0 = 0	1	0	0	0 && 1 = 0 Path7 from == holding

0	1	1	0	0	0	0 && 1 = 0	Path 8 from == holding
0	0	1	0	0	0	0 && 1 = 0	Path9 from == holding
0	0	1 && 0 = 0	0	0	1	0 && 1 = 0	Path13 from == holding
0	0	0 && 1 = 0	0	0	1	1 && 1 = 1	Path15 to == holding
0	0	0 && 1 = 0	0	0	1	0 && 1 = 0	Path 17
1	1	Not possible because from != to (line 159)					
1	-	1 && -	Not possible if invariant _holding != 0 holds				
1	-	- && 0	Not possible if the invariant all tokens in buybacks were minted before and not burned yet holds. This means that the token exists and therefore cannot be minted again				
-	1	0	Not possible because the burn function is only called to destroyBuyBacks and after removing the token from the buybacks				
-	-	0 && 0	Not possible if invariant: contains(_buybacks, tokenId) -> token.owner == holding				
1	-	-	1	Not possible because kyc.areSameUser checks that from != 0 and to != 0			
-	1	-	1				
1	-	-	-	1	Not possible if Invariant token.owner != 0		
-	1	-	-	1	Not possible because _transfer in ERC721Upgradeable ensures to != 0		
-	-	- && 0	-	1	Not possible if it is a token fix, the token it was already removed from buybacks in fixTokenOwnership		
1	-	-	-	-	1	Not possible because: isTrade = !(isMint isBurn isClaim isSameUser isTokenFix);	
-	1	-	-	-	1		
-	-	1	-	-	1		
-	-	-	1	-	1		
-	-	-	-	1	1		
0	0	0	0	0	0		
1	-	-	-	-	-	- && 1	Not possible because: isBuyBack = !(isMint isTokenFix);
-	-	-	-	1	-	- && 1	
0	-	-	-	0	-	- && 0	
-	1	-	-	-	-	1 && -	Not possible if invariant holding != 0 holds
-	-	1 && -	-	-	-	1 && -	Not possible because from != to

Control Flow Graph

The following diagram depicts all the possible paths of the function `_beforeTokenTransfer` according to the possible values of the flags (`isMint`, `isBurn`, `isClaim`, `isTrade`, `isBuyBack`, `isSameUser`, `isTokenFix`) and predicates inside `if` statements.

The unreachable paths are colored black. The reachable ones are colored according to the corresponding line in the table presented above.





Path analysis

For each possible path, we present the actual code block that it is executed:

- Path 1 - corresponds to minting. Notice on the table that the address can be the holding account.

```
if(isMint) {
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 3 - corresponds to fixing the token ownership where the new owner will be the `_holding` account. Notice on the table that `isSameUser` may be also true, which means that the `_holding` account has another KYC registered wallet and is transferring ownership from that wallet to the `_holding` account.

```
if(!isMint && isTokenFix && to==holding && !burn) {
    _checkExpectValue(Tag.brokenTokenId, tokenId, _brokenTokenId);
    _brokenTokenId = 0;
    _getToken(tokenId).owner = to;
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 5 - corresponds to fixing the token ownership, but the `newOwner` is not the `_holding` account. Notice the following situations that can happen:
 - `isClaim && isSameUser && isTokenFix` - token fix ownership for the same user. The previous owner was the `_holding` account, which means it is a transfer to another KYC registered wallet of the `_holding` account.
 - `isClaim && !isSameUser && isTokenFix` - token fix ownership with the previous owner being the `_holding` account. However, the transfer is being made to another user. This scenario should be avoided.
 - `!isClaim && isSameUser && isTokenFix` - token fix ownership for some user. The token is being transferred to another wallet of the same user. This is the intended scenario.

- `!isClaim && !isSameUser && isTokenFix` - token fix ownership for some user. The token is being transferred to another user. This scenario should be avoided since it represents a bypass to the secondary market - see [Finding Bo2](#).

```
if(!isMint && isTokenFix && to!=holding && !burn) {
    _checkExpectValue(Tag.brokenTokenId, tokenId, _brokenTokenId);
    _brokenTokenId = 0;
    _pushDiv(tokenId, to);
    _getToken(tokenId).owner = to;
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
    isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 7 - corresponds to a transfer to the same user. Notice on the table that it can represent a transfer where the from (exclusive) or to can be the `_holding` account, which means that it is a transfer between two registered wallets of the `_holding` account.

```
if(!isMint && !isTokenFix && isSameUser && !isBurn) {
    _getToken(tokenId).owner = to;
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
    isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 8 - corresponds to burning a token owned by the `_holding` account. This is the only possible scenario for burn because only tokens in the `_buybacks` are burned. Assuming invariant P12 - all tokens in `_buybacks` are owned by the `_holding` account - holds, then only tokens owned by the `_holding` account can be burned.

```
if(!isMint && !isTokenFix && !isSameUser && isClaim && isBurn) {
    _pushDiv(tokenId, to);
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
    isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 9 - Corresponds to a claim, i.e. a token transfer from the `_holding` account to the user.

```
if(!isMint && !isTokenFix && !isSameUser && isClaim && !isBurn) {
    _pushDiv(tokenId, to);
    _getToken(tokenId).owner = to;
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
    isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 13 - Corresponds to a trade between the `_holding` account and some user. Although, `from==_holding` the flag `isClaim` is not true, which means that the token is in the `buyBacks`; therefore, it has to be removed from there.

```
if(!isMint && !isTokenFix && !isSameUser && !isClaim && isTrade && from ==
holding && !buyback && !isBurn) {
    _checkTransferAuthorized();
    _checkHoldPeriod(tokenId);
    EnumerableSetUpgradeable.remove(_buyBacks, tokenId);
    _pushDiv(tokenId, from);
    _getToken(tokenId).owner = to;
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
    isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 15 - Corresponds to a `buyBack`. Since the `buyback` is also a trade, it can only be done after the `holdPeriod` ends.

```
if(!isMint && !isTokenFix && !isSameUser && !isClaim && isTrade && buyback
&& !isBurn) {
    _checkTransferAuthorized();
    _checkHoldPeriod(tokenId);
    EnumerableSetUpgradeable.add(_buyBacks, tokenId);
    _pushDiv(tokenId, from);
    _getToken(tokenId).owner = to;
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,
    isBuyBack, isSameUser, isTokenFix);
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());
    super._beforeTokenTransfer(from, to, tokenId, batchSize);
}
```

- Path 17 - Corresponds to a trade between 2 users

```
if(!isMint && !isTokenFix && !isSameUser && !isClaim && isTrade && from !=  
holding && !buyback && !isBurn) {  
    _checkTransferAuthorized();  
    _checkHoldPeriod(tokenId);  
    _pushDiv(tokenId, from);  
    _getToken(tokenId).owner = to;  
    uint256 flags = _setTransferTypeFlags(isMint, isBurn, isClaim, isTrade,  
isBuyBack, isSameUser, isTokenFix);  
    emit CRTTransfer(from, to, tokenId, flags, _msgSender());  
    super._beforeTokenTransfer(from, to, tokenId, batchSize);  
}
```