



PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For HyperCycle ShareManagerV2

09 September 2024



paladinsec.co



info@paladinsec.co

Table of Contents

Table of Contents	2
Disclaimer	3
1 Overview	4
1.1 Summary	4
1.2 Contracts Assessed	4
1.3 Findings Summary	5
1.3.1 HyperCycleShareManagerV2	6
2 Findings	8
2.1 HyperCycleShareManagerV2	8
2.1.1 Privileged Functions	10
2.1.2 Issues & Recommendations	11



Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocacy for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.


1 Overview

This report has been prepared for HyperCycle's ShareManagerV2 contract on the Ethereum network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	HyperCycle
URL	https://www.hypercycle.ai/
Platform	Ethereum
Language	Solidity
Preliminary Contracts	https://github.com/hypercycle-development/hypercycle-contracts/commit/f77caeda9766642539031061f32c9c412e901578
Resolution 1	https://github.com/hypercycle-development/hypercycle-contracts/commit/55fe8226a982d9938edf3b60d344aa62d34744ae
Resolution 2	https://github.com/hypercycle-development/hypercycle-contracts/blob/5cfcfc1cac8891e15e376a4113a230b38e07f968/contracts/ethereum/core/HyperCycleShareManagerV2.sol

1.2 Contracts Assessed

Name	Contract	Live Code Match
HyperCycleShareManagerV2	0xc5d5B9F30AA674aA210a0ec24941bAd7D8b42069	 MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● Governance	0	-	-	-
● High	5	5	-	-
● Medium	9	6	2	1
● Low	7	7	-	-
● Informational	5	3	1	1
Total	26	21	3	2

Classification of Issues

Severity	Description
● Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
● High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
● Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
● Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
● Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 HyperCycleShareManagerV2

ID	Severity	Summary	Status
01	HIGH	Users can claim more tokens from hypcSurplus than they should	✓ RESOLVED
02	HIGH	cHyPC owner can steal license token and most of the HyPC tokens that were meant for distribution after proposal ends	✓ RESOLVED
03	HIGH	Missing approvals when sending tokens leads to DOS	✓ RESOLVED
04	HIGH	shareToken owner can game the migration flow	✓ RESOLVED
05	HIGH	Changing the share manager contract leads to stuck surplus that cannot be withdrawn	✓ RESOLVED
06	MEDIUM	Hardware operator can claim above the expected operatorRevenue	ACKNOWLEDGED
07	MEDIUM	Cancellation of share migration can be frontrun	✓ RESOLVED
08	MEDIUM	The incorrect revenueDepositDelay can be set for a shareProposal during migration	✓ RESOLVED
09	MEDIUM	If operatorRevenue is changed, the amount that can be claimed by the previous rules changes as well	✓ RESOLVED
10	MEDIUM	Users that want to claim HyPC can be grieved by pendingDeposits	PARTIAL
11	MEDIUM	Operator revenue mechanism can involuntarily DoS the claiming of revenue for the users	PARTIAL
12	MEDIUM	Ambiguous ownership requirement when completing shareProposal creation	✓ RESOLVED
13	MEDIUM	Imprecise formula for computing revenue for the hardware operator results in less rewards	✓ RESOLVED
14	MEDIUM	Migrated proposals can be completed individually, resulting in loss of share tokens and revenue	✓ RESOLVED
15	LOW	The system can be spammed with proposals	✓ RESOLVED
16	LOW	Lack of minimum/maximum thresholds allows the creation of proposals with unsafe revenue delays	✓ RESOLVED
17	LOW	No minimum duration for proposal deadlines	✓ RESOLVED
18	LOW	Voting in certain cases does not close the voting and can alter the end result	✓ RESOLVED
19	LOW	Improper check inside claimHypcPortionAndSurplus()	✓ RESOLVED
20	LOW	Migration allows the creation of shareProposals that could be canceled immediately	✓ RESOLVED
21	LOW	Potential re-entrancy inside _completeProposal()	✓ RESOLVED

22	INFO	wToken holders cannot control how many tokens to deposit as their voting power	✓ RESOLVED
23	INFO	The Events.HypcClaimed event is incorrectly emitted	✓ RESOLVED
24	INFO	SELECTED_VOTATION_PERCENT should be configurable	ACKNOWLEDGED
25	INFO	Typographical issues	✓ RESOLVED
26	INFO	Insufficient validation	PARTIAL



2 Findings

2.1 HyperCycleShareManagerV2

HyperCycleShareManagerV2 is a decentralized network of AI computation nodes that offers a variety of AI services. The system is built on a foundation of multiple stakeholders, including license holders, token holders, hardware operators, and AI developers. At the core of this system is the interaction between various contracts and tokens that facilitate the operations and governance of the network.

Key Components:

- HyPC token (ERC20): The primary token of the HyperCycle network.
- cHyPC Token: A containerized version of HyPC that points to a specific license, represented as an NFT.
- HyperCycle License (NFT): A non-fungible token that grants the owner the right to offer AI services on the HyperCycle network.

The HyperCycle Share Manager

The HyperCycle Share Manager contract is designed to facilitate revenue-sharing proposals among cHyPC owners. The Share Manager holds and manages the Share Tokens, enabling decentralized governance through voting mechanisms.

Core Functionality:

1. Creating a Share Proposal

- A cHyPC owner can initiate a revenue-sharing proposal by calling the createShareProposal function.
- The cHyPC token is transferred to the Share Manager, which holds it until the proposal concludes.

- Upon the conclusion of the proposal, cHyPC is redeemed for HyPC tokens, which are then claimable by the share token holders.

2. Proposal Data

- Each Share Proposal includes detailed information:
 - **CHyPC Data:** ID, Owner, Level, Initial Revenue and Wealth Tokens.
 - **License Data:** License Number, Owner, Level, Initial Revenue and Wealth Tokens.
 - **Operator Data:** Revenue, Assigned String, Operator Address.
 - **Share Token Data:** Token Number, Revenue Deposit Delay, Revenue and Wealth Token IDs, Valid End Timestamp.
- The proposal transitions through three states: Pending, Started, and Ended.

3. Decentralized Governance

- The Share Manager integrates a DAO system where share token holders can vote on proposals to:
 - Change the hardware operator or operator revenue
 - Modify the Share Manager contract
 - Cancel an ongoing share proposal
- Voting thresholds depend on the proposal type, with a consensus requirement of 50%, 90%, or 100% of wealth tokens.

4. Proposal Migration

- The Share Manager supports the migration of Share Tokens from the Share Tokens contract to itself, allowing the Share Manager to manage the entire proposal process, including claiming HyPC tokens based on the available wealth tokens when the proposal ends.


- Migration requires the share token owner to initiate the process, and the Share Manager completes it once ownership of the share token is transferred and the proposal is in a pending state.

2.1.1 Privileged Functions

- `proposeCancelShare` [VOTER]
- `proposeNewHardwareOperatorAddress` [VOTER]
- `proposeNewHardwareOperatorRevenue` [VOTER]
- `proposeNewManager` [VOTER]
- `proposeNewDepositRevenueDelay` [VOTER]
- `vote` [VOTER]
- `cancelPendingShareProposal` [cHyPC Owner]
- `completeShareProposal` [License Owner]
- `sendRevenueToHardwareOperator` [Hardware Operator]



2.1.2 Issues & Recommendations

Issue #01	Users can claim more tokens from hypcSurplus than they should
Severity	 HIGH SEVERITY
Description	<p>Once a shareProposal is canceled, the underlying shareToken also gets canceled. Once that happens, the following flows can be executed:</p> <ul style="list-style-type: none">- claimHypcPortionAndSurplus() can be called by w/rToken holders to withdraw the surplus and HyPC balance- w/rToken owners can burn the tokens and respectively decrease the supply <p>The problem here is that wealth token holders can burn their tokens and manipulate the supply which is used in claimHypcPortionAndSurplus() to calculate the portions that should be sent out.</p> <p>More specifically, this can be used by an advantageous account when withdrawing hypcSurplus against revenue tokens. This is the problematic code snippet from claimHypcPortionAndSurplus():</p> <pre>uint256 surplusAmount = shareProposal.hypcSurplus; uint256 revenueTokenTotalSupply = managerData.hypcShareTokens.getWealthTokenTotalSupply(shareTokenNumber); uint256 totalWildRevenueTokens = (revenueTokenTotalSupply * 7) / 10; uint256 amountToRefund = (surplusAmount * userRevenueTokenBalance) / totalWildRevenueTokens; managerData.hypcShareTokens.safeTransferFrom(_msgSender(), address(this), shareProposal.shareTokenData.rTokenId, userRevenueTokenBalance, '',);</pre>

The issue is because `revenueTokenTotalSupply` is the supply of all wealth tokens (`getWealthTokenTotalSupply()` is called instead `getRevenueTokenTotalSupply()`).

Example PoC

- `wealthTotalSupply` is 1000 and they were all transferred to `shareProposal` creator at the beginning
- `revenueTotalSupply` is also 1000, but only 700 were sent to the creator in the beginning. Then he kept 350 and distributed the other 350 to other accounts

`shareProposal` is canceled and the holder of 1000 wealth tokens burns 300, leaving their balance and the `totalSupply` to be 700. This means that `revenueTokenTotalSupply` is calculated as 700, and `totalWildRevenueTokens` is 70% of it or 490.

Finally, the calculation of `amountToRefund` looks like this (assuming `surplusAmount` is 1000e18 HyPC tokens):

```
uint256 amountToRefund = (1000e18 * 350) / 490 =>
~714e18
```

The user has managed to manipulate the calculation so that it seems they are entitled to ~71% of the surplus, when in fact it should be only 50% (350/700).

Recommendation	Use <code>getRevenueTokenTotalSupply()</code> instead of <code>getWealthTokenTotalSupply()</code> when calculating <code>revenueTokenTotalSupply</code> to properly calculate the ratios.
-----------------------	---

Resolution



This issue has been resolved, however another informational issue was introduced: `cpypcData` sets `initialWealthTokens` to the same value as `revenueTokensSupply` when it should be set to 0.

Severity



- revenue: 30%

`createShareProposal()` can be called by the owner of a cHyPC token with `licenseNumber == 0` which would allow the license token owner to finish the proposal with `completeShareProposal()`.

The cHyPC owner can specify how much revenue and wealth tokens to be assigned to them and the rest will be assigned to the license owner.

For example, using the following parameters for initial values:

cHyPC owner:

- revenue: 35%; wealth: 90%

Share Manager always has:

- revenue: 30%; wealth: 0%

License owner:

- revenue: $100\% - (35\% + 30\%) = 35\%$; wealth: 10%

The proposal has status `STARTED` and operates normally for some time. We assume that `SELECTED_VOTATION_PERCENT` is 90%. This means that the initial cHyPC owner has the power to carry out proposals and votes all by themselves. They could change the share manager by using `proposeNewManager()` and they will be able to successfully carry out the execution of `_changeShareManagerContract()`.

Now, this execution will:

- Change the status of this proposal inside the current share manager to `ENDED`
- The ownership of the share token will be transferred to the new address, which could be just one of the EOAs of the cHyPC owner
- `_sendRevenueToHardwareOperator` will get called
- The 30% revenue tokens of the share manager will get transferred to this EOA

Note that now the new malicious owner of the share token can just call `cancelShareTokens()` to obtain `cHyPC` and the license token, they will be able to claim a big part of the revenue with the 30% of revenue tokens they just got from the previous share manager.

This way, they effectively steal a larger part of the revenue surplus, the full amount of `HyPC` tokens that would have been distributed according to wealth token balance ratio and the license token.

The same issue appears as well when the share is migrated—the license owner receives 0 shares while the token owner receives all the wealth supply which will leave him with full voting power.

- wealth: 0%
Ensure that the license owner is never left with 0 wealth tokens and that the votation percent for the changing of the share manager is 100%. This way, the license owner must agree before changing the share manager.

Resolution



Issue #03

Missing approvals when sending tokens leads to DOS

Severity



Description

Missing approvals in the following functions when transferring tokens leads to DOS.

`_changeShareManagerContract()` should call `managerData.hypcToken.approve(...)` between `withdrawEarnings(...)` and `depositRevenue(...)` to make sure `depositRevenue(...)` does not revert.

This has been implemented properly in `_sendRevenueToHardwareOperator()`.

Recommendation

Add the necessary approvals to ensure calls will execute properly.

Resolution



The unnecessary check against the withdrawable revenue has been removed.

Issue #04**shareToken owner can game the migration flow****Severity****Description**

A shareToken owner can use a combination of start and cancel migration calls to exploit the flow and severely affect the protocol.

Example PoC:

1. Owner of shareToken number 5 calls `startShareProposalMigration()` and creates a pending proposal with id 1: 30% of rTokens are transferred from the owner (who holds 100%) to the contract
2. The same owner calls `startShareProposalMigration()` again for the same shareToken (number 5). This creates a new proposal with id 2: 30% of the rTokens are transferred, leaving the owner with 40% of the supply.
3. The owner transfers ownership for shareToken 5 to the contract and calls `finishShareTokenMigration()` successfully, finalizing proposal with id 1
4. The owner calls `cancelShareTokenMigration()` for the proposal with id 2 and the following happens:
 - a. The ownership check passes because it compares `msg.sender` with the creator of the pending migration
 - b. The next check validates if the ownership of shareToken 5 has been transferred to the contract and since that is true (since step 3), the shareToken is transferred back to the original owner.

The end result is that the shareToken owner managed to create a proposal but then took back the shareToken. Having control over the shareToken while there is an active proposal for it opens the door to countless nasty exploits.

Recommendation

Consider adding an additional mapping that stores the shareTokens for which a migration has been started and use it to prevent creating more than one proposal migration for the same shareToken.

Resolution

Issue #05**Changing the share manager contract leads to stuck surplus that cannot be withdrawn****Severity****Description**

`_changeShareManagerContract()` does the following:

- updates the proposal status to ENDED
- calls `_sendRevenueToHardwareOperator()`

The problem lies in the fact that

`_sendRevenueToHardwareOperator()` checks if the proposal is ENDED and if that is the case, it stores the surplus in `shareProposal.hypcSurplus` instead of re-depositing it.

The surplus stored in `shareProposal.hypcSurplus` is withdrawable by `rToken` holders via `claimHypcPortionAndSurplus()`.

Withdrawing however will not be possible in this particular case because of the following validation inside `claimHypcPortionAndSurplus()`:

```
if (shareProposal.status != Types.ShareProposalStatus.ENDED
|| shareCancelled[shareProposalId] != true) {
    revert Errors.ShareProposalIsNotEnded();
}
```

Since `_changeShareManagerContract()` only sets the status to ENDED without setting the `shareCancelled` mapping, the validation above will revert and prevent surplus withdrawal.

Recommendation

Change the order of operations inside `_changeShareManagerContract()` by calling `_sendRevenueToHardwareOperator()` before setting the proposal to ENDED—this way, the revenue will get re-deposited and can be handled by the new `shareManager` contract.

Resolution

Severity



Description

The contract implements a `sendRevenueToHardwareOperator()` function which allows the configured hardware operator to call it while the proposal is active and send the revenue to themselves. This however creates an opportunity for the operator to withdraw more than they should.

Example PoC for sending Revenue to the operator:

- `claimAndWithdraw()` is called on the `shareToken` contract, which withdraws all the accumulated rewards for the 30% `rTokens` the `ShareManager` contract holds
- From the withdrawn amount, ~66% are distributed and sent to the `operatorAddress`, while the rest ~33% is re-deposited to the `shareToken` contract to be distributed among the other `rToken` holders

This allows for the following to happen:

- Some revenue is deposited and unlocked in the `shareToken` contract - `1e18`
- `operatorAddress` calls `sendRevenueToHardwareOperator()` - 66% (~66e17) are sent to the operator and 33% (~33e17) are re-deposited
- While the proposal is active, the 33% re-deposited revenue can get unlocked again (can be done by anyone), which allows `claimAndWithdraw` (called inside `sendRevenueToHardwareOperator`) to be called and withdraw those 33%, of which the operator will get 66%
- The more times this is repeated, the more fees are withdrawn

Recommendation

One solution is to store the amounts in the surplus buffer because re-deposits are always considered as revenue and will be split every time between the operator and the `rToken` holders.

Another possibility is to just ditch the operator fees mechanism and simply transfer the `rTokens` to the hardware operator, and they would be able to withdraw the revenue as a normal user.

Resolution

ACKNOWLEDGED


The client stated:

" Sending rTokens to the hardware operator is not possible, since the operator can be changed at any time, and the corresponding rTokens would not be able to be taken back from the old specified operator.

Putting the redeposits into the surplus would delay this revenue from being sent out from the contract since surpluses do not get sent out until the end of the share (canceled or migrated). Since canceling shares and migrating them is an uncommon event for a share, it doesn't make sense to lock the funds inside the share until then, so that's why redepositing revenue makes more sense. The math of this (the hardware operator getting a little more than the hardware operator percentage, is expected."



Severity

 MEDIUM SEVERITY

Description

A user can migrate shares created outside the Share Manager to the Share Manager. The purpose of this is to allow the Share Manager to manage the user's shares and handle the claiming process of revenue.

The migration process is as follows:

1. The user calls `startShareProposalMigration()`, registering the proposal under `shareProposalId` with the status set to `PENDING`
2. The user sends the share token to the Share Manager
3. Any user can call `finishShareTokenMigration()` to complete the share migration

The issue arises from the fact that anyone can call the function to finish the token migration. This creates a potential problem: if the user sends the share ownership to the contract then decided to actually cancel the migration, a malicious actor could monitor the transaction in the mempool and call the `finish` function first which would mark the share migration as complete, thus preventing the cancellation.

Recommendation

Consider guarding the `finish` function to be called only by the `tokenOwner`.

If this function should be publicly callable then consider adding some type of timelock that starts counting after the migration has started. For the duration of that timelock, make sure that only the owner can call `finishShareTokenMigration()`. This would ensure the owner will have enough time to resolve the migration. After the timelock expires, the `finishShareTokenMigration()` can be unlocked to be called by anyone.

Resolution

 RESOLVED

Issue #08**The incorrect revenueDepositDelay can be set for a shareProposal during migration****Severity****Description**

startShareProposalMigration() is used to create a shareProposal by an account that already owns a shareToken. The creation happens in 2 stages:

- startShareProposalMigration() creates the data for the new shareProposal based on the existing shareToken
- finishShareTokenMigration() finalizes the creation by transferring the ownership of shareToken to HyperCycleShareTokensV2

The issue is that startShareProposalMigration sets revenueDepositDelay inside _shareProposal.shareTokenData to the current value set for that shareToken.

Since the ownership is transferred later when calling finishShareTokenMigration(), the owner of shareToken can call changePendingRevenueDelay() on the shareToken contract and update the value after migration started but before finalizing it.



After that, they can call finishShareTokenMigration() which assumes revenueDepositDelay has not been changed and as a result the shareProposal is created with an invalid revenueDepositDelay value.

Recommendation

Similar to how operatorAssignedString is handled inside finishShareTokenMigration(), add the following logic to make sure the delay is updated to the value that was set when migration was started:

```
managerData.hypcShareTokens.changePendingRevenueDelay(  
    shareTokenNumber,  
    shareProposal.shareTokenData.revenueDepositDelay  
);
```

Resolution

Issue #09	If operatorRevenue is changed, the amount that can be claimed by the previous rules changes as well
Severity	 MEDIUM SEVERITY
Description	<p>When a vote for changing hardware operator revenue passes, <code>_changeHardwareOperatorRevenue</code> gets called and changes the revenue ratio that the operator will get from then on. However, <code>_sendRevenueToHardwareOperator</code> is not called within <code>_changeHardwareOperatorRevenue</code> before changing the revenue with the new value. This means that the operator will receive their previously generated rewards in the new revenue ratio instead of with the old one.</p> <p>This could be unfair since the operator could initially work for a 20% cut and would have gotten X but after the vote, their cut could be lowered to 10%. Because they did not claim his revenue before the vote, they will receive $X / 2$.</p>
Recommendation	Consider calling <code>_sendRevenueToHardwareOperator</code> inside <code>_changeHardwareOperatorRevenue</code> before writing the new value in storage.
Resolution	 RESOLVED

Issue #10**Users that want to claim HyPC can be grieved by pendingDeposits****Severity** MEDIUM SEVERITY**Description**

HyperCycleShareTokensV2.depositRevenue() can be called by anybody while the proposal is still active. By not having a minimum required amount for deposit allows for a malicious user to make thousands of 1 wei deposits which in return would force users that want to use HyperCycleShareManagerV2.claimHypcPortionAndSurplus() to call unlockRevenue() for each such pending deposit which could be expensive in terms of gas.

Recommendation

Unfortunately as HyperCycleShareTokensV2 is already deployed, there is not an easy fix other than redeploying this contract.

Resolution PARTIALLY RESOLVED

The issue has been partially resolved but a risk of DoS still exists if anyone triggers claim revenue for a user which will make the call to claimHypcPortionAndSurplus to revert by in the following:

```
if
(managerData.hypcShareTokens.lastShareClaimRevenue(shareTokenNumber, _msgSender())) != revenueDeposited) {
```

In the third resolution commit, the revenue check was also moved only if overridePendingDeposits is false. However, this issue is not completely resolved because the check of the withdrawable amount is still always done.

Description

The operator revenue formula is adjustable to be between 0-30% of the revenue, using the rTokens of the share. To accommodate this adjustability, 30% of the revenue tokens are held in the Share Manager.

When the Hardware Operator wants to claim their revenue, `sendRevenueToHardwareOperator` is called. Within this function, the following algorithm is defined:

```
uint256 hypcSurplus = amountReceived -
hardwareOperatorRevenue;
if ( shareProposal.status != Types.ShareProposalStatus.ENDED
) {
    managerData.hypcToken.approve(address
(managerData.hypcShareTokens), hypcSurplus);
    managerData.hypcShareTokens.depositRevenue(
shareProposal.shareTokenData.shareTokenNumber, hypcSurplus);
} else {
    // @dev Add this to the hypcSurplus, that will be claimed
on a per-user basis.
    shareProposal.hypcSurplus += hypcSurplus;
}
```

This states that if the share is not yet ended, the surplus between the 30% revenue and the operator revenue is sent to be distributed to the users by calling `depositRevenue` in the `HyperCycleShareTokensV2` which adds a new pending deposit.

After the share proposal has ended, users can claim their revenue by calling `claimHypcPortionAndSurplus`. Within this function we have the following check:

```
if
(managerData.hypcShareTokens.getPendingDepositsLength(shareT
okenNumber) != 0) {
    revert Errors.MustUnlockRevenueBeforeClaimingSurplus();
}
```

Which states that in order to claim the revenue, no pending deposits should be active.

In order to clean the deposits, an open function must be called to unlock the revenue after `revenueDepositDelay` has passed on that deposit entry.

As we can see, due to these constraints, a malicious hardware operator or a hardware operator that erroneously calls `sendRevenueToHardwareOperator` could create a bunch of pending deposits that delay the claiming for the users.

Recommendation A cleaner approach would be to ditch the 30% lockup that is reserved for adjusted share of the revenue of the hardware operators and just transfer the rTokens to the hardware operator, ultimately letting the operator claim the revenue as a normal user. This will force the operator to not receive revenue while the share proposal is active.

Another solution would be to add a timelock to the `sendRevenueToHardwareOperator` to be called once every X amount of time. This still will produce pending deposits for a share proposal but at least will minimize the risk of DoS.

Resolution

 PARTIALLY RESOLVED

Same as Issue #10.



Issue #12**Ambiguous ownership requirement when completing shareProposal creation****Severity** MEDIUM SEVERITY**Description**

When calling `createShareProposal()`, the caller can choose to complete the creation in the same transaction by providing a `licenceNumber` or doing it at a later stage by calling `completeShareProposal()`.

`completeShareProposal()` implements a `onlyLicenseOwner` modifier that ensures only the actual owner of the license can provide it to the protocol and complete the creation of `shareProposal`.

However, there is no such requirement within `createShareProposal()`. The caller is required to only be the owner of the `chYPC` token. This means that they can be approved but not the real owner of the `licenceNumber`. Since `msg.sender` is set as `tokenOwner` of the license for the `shareProposal`, the approved address would be considered as owner in that case

This discrepancy in both functions creates ambiguity and can lead to unexpected consequences.

For example, in the case of `createShareProposal()`, if an approved address is set as the license `tokenOwner`, the revenue and wealth tokens after share creation will get sent to that address. Also the license token will get returned to it once `shareProposal` is canceled.

Recommendation

Consider assigning a license token (ERC721) approved address as `tokenOwner` is an acceptable and expected scenario and synchronize the logic across functions to either allow or prevent it.

This ensures the logic is consistent and mitigates unexpected outcomes.

Resolution RESOLVED

Severity

 MEDIUM SEVERITY

Description

The operator revenue formula is adjustable between 0-30% of the revenue using the rTokens of the share. To accommodate this adjustability, 30% of the revenue tokens are held in the Share Manager.

The initial percentage that should be distributed to the operator is computed as follows:

$$(\text{wealthTokensSupply} \ll 1) / 10$$

which translates to:

$$2 * \text{wealthTokensSupply} / 10, \text{ resulting in } 20\%.$$

When calculating the revenue to be sent to the operator, the contract calls `claimAndWithdraw` of `HyperCycleShareTokensV2`, which allocates a percentage of the revenue based on the 30% holdings of the revenue tokens.

Out of the 30%, two-thirds are sent to the operator (since they should receive 20% of the total revenue) and 10% is returned to be distributed to the users.

While this approach theoretically works as expected, in practice, it results in less revenue for the operators due to rounding down both when the operator revenue is initially set and when calculating the percentage of the amount that should be distributed:

```
(amountReceived * shareProposal.operatorData.operatorRevenue * 10) / ((1 << shareProposal.chypcData.tokenLevel) * 3);
```

Example:

- $\text{wealthTokensSupply} = 2^{19}$
- Operator revenue is set to $(2^{19} \ll 1) / 10 = 104,857$, which is already less than 20% (19.99%).

If the reward received for 30% is 1,000,000 (1e6), the amount received by the operator would be:

$$(1e6 * 104,857 * 10) / ((2^{19}) * 3) = 666,662$$

Assuming that 1e6 represents 30%, the full 100% would be 3,333,333, and 20% of 3,333,333 would be 666,666, not 666,662.

Recommendation We suggest that the actual percentage of revenue for the operator should be calculated using a percentage scaling factor, which is a common algorithm used within DeFi. For example, if we consider 30% as the maximum reward (representing 100%), the operator's reward should be scaled relative to that, rather than directly to the whole 100%.

We propose the following algorithm:

operatorRevenue should hold a value between 0 and 100,000 (where 100,000 represents 100% of the 30% maximum reward).

When calculating the revenue, the following formula should be used:

$$(\text{amountReceived} * \text{shareProposal.operatorData.operatorRevenue}) / 100_000$$

This method ensures that the calculation always approximates to the nearest decimal.

Resolution



Severity



Description

A user can potentially complete a share proposal migration with a new license, leading to:

- Overriding the original proposal migration that was initiated.
- Loss of revenue tokens that were already sent during the migration start.

For example:

1. A user starts migration for share token number 1.
2. Revenue tokens are sent to the Share Manager in `startShareProposalMigration`.
3. The proposal status is set to PENDING:
`_shareProposals[shareProposalId].status = Types.ShareProposalStatus.PENDING.`
4. The user cancels the share token by calling `HyperCycleShareTokensV2.cancelShareTokens()` and receives the `chypcV2` with ID 5.
5. The user then initiates a new proposal in the Share Manager with `createShareProposal`, and `chypcV2` with ID 5 is transferred back to the Share Manager.
6. A malicious License Owner completes the old share migration proposal, since the `chypcV2` with ID 5 is still present in the Share Manager contract. This results in:
 - `shareProposal.shareTokenData` being populated with a new share token number (2 in this case, as the share token number has increased).
 - New revenue/wealth tokens being created, but the quantities for the initial revenue and wealth tokens are based on the values from the start of the migration function, which may be incorrect.
 - Loss of old revenue tokens that were transferred at the start of the migration.

Recommendation Consider separating the processes of completing a share proposal created via migration and creating a new share proposal. These two actions should be handled as distinct workflows.

To address this, a new status, `STARTED_MIGRATION` could be introduced. The `cancel` and `finish` functions should then check against this status rather than the `PENDING` status. This would ensure that the migration process is not inadvertently overridden or interrupted.



Another potential solution is to take custody of the share token directly at the start of the migration. By doing so, the owner would be unable to perform any actions on the share during the migration process, preventing any interference or potential issues.

Resolution

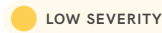


The migration process has its own status now.



Issue #15	The system can be spammed with proposals
Severity	 LOW SEVERITY
Description	<p>There is nothing in place to prevent an account with votePower to create an unlimited amount of proposals from a particular type. For example, proposeCancelShare() requires that the caller only needs to have vote power and a valid deadline. This means they can push as many proposals as they want into the _votations array.</p> <p>This does not provide any value to the protocol and should not be allowed.</p>
Recommendation	Consider using a mapping that stores the deadline from the last created proposal of this type for the account and ensure that a new proposal can be created only if the current one has expired.
Resolution	 RESOLVED



Issue #16**Lack of minimum/maximum thresholds allows the creation of proposals with unsafe revenue delays****Severity****Description**

When creating a proposal, the creator defines the `revenueDepositDelay` parameter which determines the time duration for which the revenue deposited inside `HyperCycleShareTokensV2` will remain locked until they can be withdrawn by revenue token holders.

Since there is no restriction on the maximum allowed delay, a `shareProposal` creator can define a very high value, making revenue deposit finalization practically impossible.

This in turn can have unintended consequences such as:

- No revenue getting finalized and distributed to the `hardwareOperator` when calling `_sendRevenueToHardwareOperator()`
- DOS on `claimHypcPortionAndSurplus()`, which is responsible for allowing revenue and wealth token holders to withdraw revenues and HyPC tokens once a `shareProposal` is canceled. The DOS occurs due to the following check:



```
if
(managerData.hypcShareTokens.getPendingDepositsLength(s
hareTokenNumber) != 0) {
    revert
    Errors.MustUnlockRevenueBeforeClaimingSurplus();
}
```

Recommendation


Define minimum and maximum limits for `revenueDepositDelay` that will ensure delays can only be set within reasonable boundaries. The limits should be implemented in the following functions:

- `createShareProposal()`
- `startShareProposalMigration()`
- `proposeNewDepositRevenueDelay()`

Resolution

Issue #17	No minimum duration for proposal deadlines
Severity	 LOW SEVERITY
Description	There is no restriction within <code>validProposedDeadline()</code> that prevents users from spamming proposals that have a duration of 1 second. Such proposals provide no real value to the protocol and should not be allowed.
Recommendation	Inside <code>validProposedDeadline</code> make sure to validate that the deadline parameter is not below a minimum acceptable value.
Resolution	 RESOLVED



Issue #18**Voting in certain cases does not close the voting and can alter the end result****Severity** LOW SEVERITY**Description**

When a user votes and the required amount of votes is reached, `votation.amountReached` will be marked as `true`. This is true for almost all cases except for `CANCEL_SHARE` and `CHANGE_MANAGER_CONTRACT` proposals.

Once the votes *against* become the required ratio of all votes (`SELECTED_VOTATION_PERCENT`) then `votation.amountReached` will not be marked as `true`.

By having in mind that `SELECTED_VOTATION_PERCENT` can be either 90% or 100%, this is not such a problem as there cannot be any impact. The vote will simply expire and nothing will happen.

If in the future `SELECTED_VOTATION_PERCENT` is set to $\leq 50\%$, then the votes (for / against) that reach the `SELECTED_VOTATION_PERCENT` first will determine the final result from the voting.

Since votes against can not mark `votation.amountReached` as `true`, there is no way for the negative votes to win. In this case, even if votes *against* reach 50% and after that the votes *for* reach the other 50%, the vote will carry the action of the vote *for*.

Recommendation

For consistency and future proof consider marking `votation.amountReached` as `true` whenever the desired ratio of for / against votes are reached.

Resolution RESOLVED

Issue #19**Improper check inside `claimHypcPortionAndSurplus()`****Severity** LOW SEVERITY**Description**

The following check inside the function is not defined properly:

```
if (userWealthTokenBalance == 0 && userVotePower +  
    userRevenueTokenBalance == 0) {  
    revert Errors.NoWealthOrRevenueTokensAvailable();  
}
```

The check is intended to validate two things: wealth and revenue token balances. Since `userVotePower` is the amount of wealth tokens the user has deposited in the contract, the correct check should be like this:

```
if (userRevenueTokenBalance == 0 && userVotePower +  
    userWealthTokenBalance == 0) {  
    revert Errors.NoWealthOrRevenueTokensAvailable();  
}
```

The deposited wealth tokens should be added to the ones in user balance, which will calculate the total wealth token balance of the user.

Recommendation

Consider updating the check above.

Resolution RESOLVED

Issue #20**Migration allows the creation of shareProposals that could be canceled immediately****Severity** LOW SEVERITY**Description**

`_cancelShareProposal()` allows a `shareProposal` to be canceled only if the `validEndTimestamp` is lower than `block.timestamp` (has expired). The value of `validEndTimestamp` is set within `startShareProposalMigration()` like this:

```
validEndTimestamp:  
managerData.hypcShareTokens.getShareStartTime(shareTokenNumber) + 1 days
```

It takes the time of creation of the `shareToken` and adds 1 day to it. So if for example the token was created two days ago, then the condition inside `_cancelShareProposal()` would be immediately valid.



Recommendation



Consider if the creation of immediately cancellable `shareProposals` is an accepted scenario.

If not then there are 2 possible approaches:

- Revert if `shareToken` was created more than 1 day ago.
- Calculate `validEndTimestamp` similar to `createShareProposal()` by using `block.timestamp + 1 days`

Resolution RESOLVED

Issue #21	Potential re-entrancy inside <code>_completeProposal()</code>
Severity	 LOW SEVERITY
Description	<p><code>_completeProposal()</code> executes two consecutive calls at the end to <code>_sendRevenueAndWealthTokens()</code> which in turn uses <code>safeTransfer</code> to send out ERC1155 tokens.</p> <p>Although the calls happen at the end, there still exists a slight issue with re-entrancy—since the first call that sends the tokens to the license owner creates a condition where he can execute some logic and be at an advantageous position as he received the tokens while the cHyPC owner has not yet received them.</p> <p>No serious issues have been detected to come from that, hence we report this as low. Still it is a good practice to minimize such possibilities as much as possible.</p>
Recommendation	We advise to have a non-reentrant modifier on all public functions to avoid any unintended situations. Unfortunately, for already deployed contracts (not in scope) there is nothing that can be done other than analyzing if this issue poses a risk.
Resolution	 RESOLVED

Issue #22	wToken holders cannot control how many tokens to deposit as their voting power
Severity	 INFORMATIONAL
Description	<p>increaseVotePower () always transfers the full balance of wTokens for an account. It is a good idea to allow the account to specify how much tokens it would like to deposit—for example it might want to deposit 50%.</p> <p>This will also allow a minimum threshold for deposit to be defined.</p>
Recommendation	Consider adding an additional parameter to the function which defines how ,amu tokens the vote power will get increased. Also a minimum amount should be defined for that parameter.
Resolution	 RESOLVED



Issue #23**The Events.HypcClaimed event is incorrectly emitted****Severity** INFORMATIONAL**Description**

After a proposal is finished and marked as ENDED, the user can claim the HyPC tokens in exchange of wealth and revenue tokens. The claiming is done twice in the same function, depending if the user has both revenue tokens and wealth tokens.

```
if ( userRevenueTokenBalance > 0 ) {  
    ...  
    managerData.hypcToken.transfer(  
        _msgSender(),  
        amountToRefund  
    );  
}  
...  
if ( userWealthTokenBalance + userVotePower > 0 ) {  
    managerData.hypcToken.transfer(  
        _msgSender(),  
        userTransferAmount  
    );  
  
    emit Events.HypcClaimed(shareProposalId,  
        userTransferAmount);  
}
```

As seen above, the event that specifies how much was claimed is emitted only if wealth tokens are available when in fact it should be emitted with the amount of both HyPC from revenue and wealth tokens conversion.

Recommendation

Consider transferring the HyPC tokens only once by accumulating the amounts from both revenue and wealth tokens, then consider emitting the event with that sum.

Resolution RESOLVED

Severity

 INFORMATIONAL

Description


The `SELECTED_VOTATION_PERCENT` is declared as public, but it is being used as an immutable variable since it is set only once in the constructor. This is an important variable that can currently be set to either 90% or 100% and determines the required quorum for executing the two most critical proposal types: `CANCEL_SHARE` and `CHANGE_MANAGER_CONTRACT`.

Since 90% and 100% can be a bit hard to achieve, it would be beneficial to the protocol to have some flexibility over that value. For example, after a bit of trial and error it might turn out that it is better to use 80% or 70%. There is no way for the protocol to react in case such necessity arises.

Recommendation

Consider adding a setter function that would allow `SELECTED_VOTATION_PERCENT` to be modified in case that is needed.

Resolution

 ACKNOWLEDGED

The team has decided not to add a setter function due to the contract size limit.

Severity

 INFORMATIONAL

Description

Consider renaming the public `sharesProposalsAmount` variable to `sharesProposalCount` to more accurately describe its purpose, which is an incrementing counter for generating proposal ids.

The developer's comment above `createShareProposal()` describes the parameters that get decoded inside it. Among those parameters is `chypcExist`. Since it is not used in the function, it can be removed it from the description.

On the other hand, there is no description for `licenseNumber`, so consider adding it in.

Within `startShareProposalMigration()`, the result of calling `getRevenueTokenTotalSupply()` is stored in a variable named `wealthTokensSupply`. This is not very accurate since this is the supply of revenue tokens. Consider renaming it to `revenueTokensSupply`.

The comment in `_endShareProposal` states: "It will send the license only if it isn't splittable". However such validation is not present in the function implementation.

There is a typographical error in the developer comments of `_endShareProposal()`: *Private function to end the finish the share proposal* be *Private function to end the share proposal*.

`HyperCycleShareManagerV2.sol` constructor has a payable modifier. Since the contract does not use native currency in any way, it should be removed.

Recommendation

Consider fixing the typographical issues.

Resolution

 RESOLVED

Description

Within `cancelPendingShareProposal()`, add a check that restricts the function from getting called with migrated shareProposals that are pending. This will ensure it returns early instead of wasting gas and reverting later in `_endShareProposal()`.

The following check can be added:

```
if
  (_shareProposals[shareProposalId].shareTokenData.shareTokenNumber != 0)
  revert Errors.ShareProposalIsMigrated();
```

The same recommendation applies to `completeShareProposal()`—it should revert early if it is a migrated pending shareProposal.

Within `completeShareProposal()`, validate that `licenseNumber` is not 0.

Within `_completeProposal()`, validate that at least `initialRevenueTokens` or `initialWealthTokens` is not 0 before calling `_sendRevenueAndWealthTokens()`.

Within `startShareProposalMigration()`, validate that `shareTokenNumber` is not 0.

Check that `operatorAssignedString` is not empty, just as in `createShareProposal()`:

```
if (bytes(operatorAssignedString).length == 0) revert
  Errors.InvalidProposedString();
```

Within `cancelShareTokenMigration()`, validate that `shareProposalId` is not 0. Also validate that `shareTokenNumber` is not 0 to verify this is a migrated shareProposal.

Within `finishShareTokenMigration()`, add a top-level check that the `shareToken` is still active, similar to how this is checked inside `startShareProposalMigration()`. Also validate that `shareProposalId` is not 0.

—

Check that `operatorAssignedString` is not empty in `proposeNewHardwareOperatorAddress()`, just as in `createShareProposal()`:

```
if (bytes(operatorAssignedString).length == 0) revert  
Errors.InvalidProposedString();
```

—

Proposal creation functions do not validate if the share proposal is already ended. Consider using the `proposalActive` modifier.

Recommendation	Consider implementing the validation recommendations.
-----------------------	---

Resolution

 PARTIALLY RESOLVED

Some issues could not be resolved as the contract size limit was reached.





PALADIN
BLOCKCHAIN SECURITY