



PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For HyperCycle Swap V2

31 October 2023



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	5
1.3 Findings Summary	6
1.3.1 HyperCycleSwapV2	7
2 Findings	8
2.1 HyperCycleSwapV2	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 Swap's v2 contracts 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/blob/693a115ea048e9f38bf46b4583a79ad7952979a6/contracts/ethereum/core/HyperCycleSwapV2.sol
Resolution 1	https://github.com/hypercycle-development/hypercycle-contracts/blob/f1f3d0b74e43e02ee1ffc2f803bd03167e3086f8/contracts/ethereum/core/HyperCycleSwapV2.sol
Resolution 2	https://github.com/hypercycle-development/hypercycle-contracts/blob/2dd07868a84352bd205cbd2d05de30adb21de062/contracts/ethereum/core/HyperCycleSwapV2.sol
Resolution 3	https://github.com/hypercycle-development/hypercycle-contracts/blob/3955e27e9f85008d792527f5f9fbfa96bd2543e6/contracts/ethereum/core/HyperCycleSwapV2.sol

1.2 Contracts Assessed

Name	Contract	Live Code Match
HyperCycleSwapV2	0x21468e63abF3783020750F7b2e57d4B34aFAfba6	UNMATCHED

The team made some changes after the audit was completed, and we are unable to verify the impact of the changes.

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● Governance	0	-	-	-
● High	1	1	-	-
● Medium	1	1	-	-
● Low	5	3	-	2
● Informational	4	2	-	2
Total	11	7	-	4

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 HyperCycleSwapV2

ID	Severity	Summary	Status
01	HIGH	Splitting of tokens will result in duplicate tokenIDs	✓ RESOLVED
02	MEDIUM	Potential manipulation of the license value	✓ RESOLVED
03	LOW	Swaps can potentially be DoS'ed	✓ RESOLVED
04	LOW	Violation of the checks-effects-interactions pattern	✓ RESOLVED
05	LOW	Token can only have number or string assigned	ACKNOWLEDGED
06	LOW	Different owners can assign their tokenIDs to the same assignedNumber	ACKNOWLEDGED
07	LOW	String is not backed by values	✓ RESOLVED
08	INFO	Redeeming a NFT can make a level exceed the LEVEL_LIMIT number of NFT	ACKNOWLEDGED
09	INFO	Token split can be DoS'ed	ACKNOWLEDGED
10	INFO	The tree is hard to query	✓ RESOLVED
11	INFO	Gas optimizations	✓ RESOLVED

2 Findings

2.1 HyperCycleSwapV2

HyperCycleSwapV2 stands as a pioneering contract within the HyperCycle token ecosystem, offering a multi-layered NFT licensing framework.

At its core, NFTs are stratified into distinct levels, starting from the foundational level 19 down to the most accessible level 10. The contract entrusts the sole authority to create these NFTs for level 19 to the owner.

As new tokens emerge, they sequentially advance from an initial root number, ensuring a systematic increase, starting from zero.

One of the contract's distinguishing features is the capacity to split tokens. This means tokens from elevated levels can be strategically distributed to spawn multiple tokens in the subsequent level. For example, a single token from level 19 can birth two at level 18, or even leap to create four tokens at level 17.

For all levels below the root level, the maximum number of different tokenIds is set to 16 to ensure availability of tokens at different levels. In terms of acquisition, each NFT's price directly correlates with its designated level. As such, a token from level 19 fetches a cost of 2^{19} HyperCycle tokens, with the pricing scale adjusting correspondingly for the other levels.

This structured pricing framework not only ensures a wide spectrum of affordability but also beckons a diverse user demographic.

To foster a consistent flow of availability across all layers, the onus lies on the contract owner to consistently replenish the pool of root tokens. This ensures potential users are continually presented with varied purchasing avenues. After

users have made a purchase, they gain custodianship over their token ID, safeguarding it from any future splits.

What's more, the owner can further embellish these token IDs by assigning unique string and number denominations, with the latter being pivotal for layering value onto a chosen target number by leveraging the formula (2^{level}). We assume that the assigned number towards a tokenId is directly owned by a user, therefore, since the user owns this specific license number, one can simply purchase tokens and aggregate their total value towards this license number, which can be fetched by third party contracts via the `getAssignmentTargetNumber` function. It has to be noted that the number aggregates the values of all tokens backing it, but the string does not.

However, the versatility does not end there. Users can redeem their tokens, effectively receiving an equal amount of their initially locked HyperCycle tokens. This act simultaneously resets the token's level and erases its previously attributed values.

In terms of security, the contract is fortified with countermeasures against any adversarial intentions. By placing a cap on each non-root level and with the inherent limitation of the HyperCycle token supply, monopolistic tendencies are effectively curtailed.

Concisely, the `HyperCycleSwapV2` contract is an intricate dance of tiered NFT licensing, interwoven with the very fabric of the HyperCycle token.

Through its design, it champions diversity in accessibility, empowers users to craft their unique value-laden license numbers, and seamlessly integrates robust security features, making it a beacon in modern blockchain contract design.

2.1.1 Privileged Functions

- `transferOwnership`
- `renounceOwnership`
- `addRootTokens`



2.1.2 Issues & Recommendations

Issue #01

Splitting of tokens will result in duplicate tokenIDs

Severity

 HIGH SEVERITY

Description

Due to how new tokenIDs are calculated within the `splitHeldTokens` function:

```
uint256 tokensToCreate = 2 ** skipLevels;

for (_i = 0; _i < tokensToCreate; _i++) {
    levels[level - skipLevels].push(tokenNumber *
tokensToCreate + _i);
    nftTokens[tokenNumber * tokensToCreate + _i] =
TokenData('', 0, level - skipLevels, false);
}
```

A scenario can encounter where level 19 and level 18 can have the same tokenIDs. This will presumably break the contract logic since it is not possible to mint the same tokenId twice.

Consider the following scenario:

1) The contract owner adds 4 root tokens.

With the initial value of `_rootData.currentRootNumber` being 0 (default) and given that `tokens = 4`, the following tokens will be added to level 19: tokenId 0 tokenId 1 tokenId 2 tokenId 3.

Then `_rootData.currentRootNumber` is incremented by 4, making its new value 4.

2) User calls `splitHeldToken` with level 19, index 1, and 1 level to skip:

Since `skipLevels = 1`, `tokensToCreate = 2`, two new tokens will be created in level 18.

Given that `level = 19` and `index = 1`, `tokenNumber` will be 1 as this corresponds to tokenId 1 in level 19.

The token with tokenId 1 is then removed from level 19.

Therefore, the token with tokenId 1 will be replaced by the token with the last tokenId in level 19, which is tokenId 3. After this replacement, tokenId 3 is removed from the end of the array. At this point, the remaining tokenIds in level 19 are: tokenId 0 tokenId 3 tokenId 2.

Now, two new tokens will be created in level18. The formula $\text{tokenNumber} * \text{tokensToCreate} + _i$ with the given values will result in:

For $_i = 0$: $1 * 2 + 0 = 2$

For $_i = 1$: $1 * 2 + 1 = 3$

Therefore, the new tokenIds created in level18 are: tokenId 2 tokenId 3.

To summarize —

In level19: tokenId0, tokenId3, tokenId2

In level18: tokenId2, tokenId3

tokenId2 and tokenId3 will get a fresh level assigned:

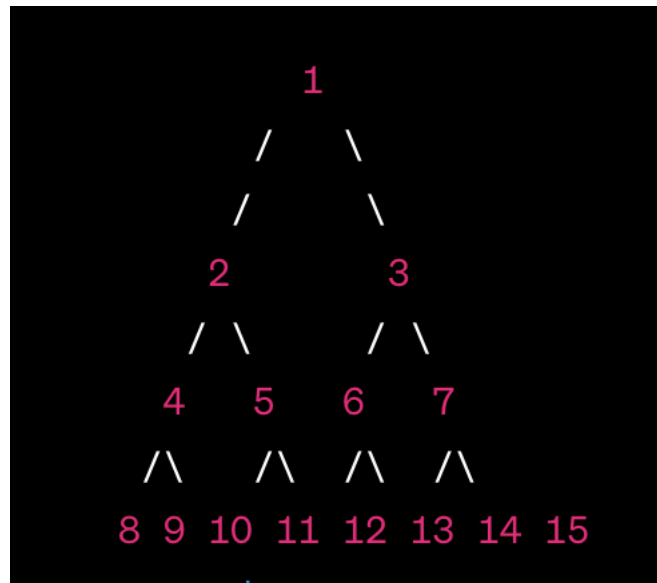
```
for (_i = 0; _i < tokensToCreate; _i++) {  
    levels[level - skipLevels].push(tokenNumber *  
tokensToCreate + _i);  
    nftTokens[tokenNumber * tokensToCreate + _i] =  
TokenData('', 0, level - skipLevels, false);  
}
```

This will result in a purchase of tokenId2 and tokenId3 within level 19 but actually receiving a token with the lower level, as the assignment is just overridden.

Another example would be if a user already purchased tokenId2 and another user splits down tokenId1 for 1 level, the user with tokenId2 would essentially only receive 2^{18} tokens when redeeming instead of 2^{19} , which was the initial purchase price.

From a financial perspective, this issue can ultimately lead to a loss of HyperCycle tokens.

The background of this issue lies within the fact that these are intentionally selected small root levels, resulting in an edge-case where the binary tree will look as follows:



Due to the small initial root, the consecutive levels can potentially include such a value which can also be part of the initial root.

Recommendation The simplest way would be to just do `currentRootNumber + 1` and then incrementing `currentRootNumber`, which would result in a slight refactoring of `addRootTokens` but would be probably the most bullet-proof method.

It is important to note that for any provided solution, it is mandatory to provide proper testing scenarios, ensuring that this edge-case is prevented. This includes stateful fuzzing with a check that all inputs result in a unique output id.

Resolution



No change was made, however the team added the following comment—

This is not an issue due to the design of the token range. Using the binary tree, all node numbers underneath a given root node are unique amongst all other root nodes at the same level (the binary string of the numbers in the tree corresponds to the path taken from the root node down to the given node).

For a token range to be valid it must be within the range of subsequent powers of 2. In our case, $[2^{26}, 2^{27})$, though we only use the first 4096 in this range. If numbers on different levels are used, then the binary string of one of the numbers could be the prefix of the other (for example, $\text{bin}(1) = "1"$ and $\text{bin}(2) = "10"$), which means that the token with the shorter prefix (ie: $\text{bin}(1) = "1"$) could be eventually split to generate the other number (in this case, 1 could would be split into "10" and "11", with $\text{bin}(2) = "10"$), causing a uniqueness issue.

—
This in fact reflects our observations that the tree will work if the initial root tokens are from a large enough number.

Issue #02 Potential manipulation of the license value	
Severity	 MEDIUM SEVERITY
Description	<p>Within the assignNumber function, the value of a specific NFT can be assigned to a license number.</p> <p>We are not aware of how this license is used, but if there is any possibility in third-party contracts to reenter into a function after a license value has been fetched, this could be abused to reenter into the assignNumber function, then assign a new number for the same tokenId which was already assigned to a license number and assign a new license number, potentially artificially increasing the value of multiple licenses.</p>
Recommendation	<p>Since this issue is very far-fetched but has a likely large impact, it is considered as medium risk. However, depending on the scenario of how the license is used in third-party contracts, this can result in damage.</p> <p>Consider adding a simple modifier towards this function which prevents it from be called twice in the same block by the same address.</p>
Resolution	 RESOLVED <p>Such a modifier has been added.</p>

Issue #03 Swaps can potentially be DoS'ed	
Severity	 LOW SEVERITY
Description	Whenever users attempt to swap tokens, the token at the desired index is fetched. However, a malicious user can frontrun the swap such that the desired level has no tokens left to be swapped.
Recommendation	Consider always ensuring that sufficient root tokens are added to prevent such a state.
Resolution	 RESOLVED <p>The team assured us that there will be sufficient root tokens available to prevent this state.</p>

Issue #04**Violation of the checks-effects-interactions pattern****Severity** LOW SEVERITY**Description**

Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes).

L354

```
function _swapV2(uint256 level, uint256 index) internal {  
    //accept the HyPC here and deposit into the contract.  
    uint256 amount = (2 ** level) * SIX_DECIMALS;  
    _hpcToken.transferFrom(msg.sender, address(this),  
amount);  
    totalLocked += 2 ** level;
```

L387

```
function redeem(uint256 tokenNumber) external  
isOwnerOf(tokenNumber) {  
    _unassign(tokenNumber);  
  
    uint256 level = nftTokens[tokenNumber].level;  
    levels[level].push(tokenNumber);  
    _updateNFTArray(level);  
    totalLocked -= 2 ** level;  
  
    transferFrom(msg.sender, address(this), tokenNumber);  
  
    uint256 amount = (2 ** level) * SIX_DECIMALS;  
    _hpcToken.transfer(msg.sender, amount);
```

Recommendation

Consider following the CEI pattern.

Resolution RESOLVED

Issue #05**Token can only have number or string assigned****Severity** LOW SEVERITY**Description**

The only entry points to assign something to a token are:

- a) `assignNumber (number)`
- b) `assignString (string)`
- c) `assign (string)`

Whenever one of these three functions is called, a full un-assigning happens, which basically means there is no possibility to have a number and a string assigned.

Recommendation

Consider if that is desired, if not, consider refactoring this logic to allow assigning both attributes.

Resolution ACKNOWLEDGED

The string is only used for backwards compatibility, therefore it is in fact desired to only have values assigned to a license number.



Description

Whenever token owners assign numbers to their tokenID, this number will accumulate the value of all tokens which have this number assigned:

```
if (targetNumber > 0) {  
    TokenData storage _targetToken = nftTokens[tokenNumber];  
    _targetToken.assignedNumber = targetNumber;  
    assignedNumbers[targetNumber] += 2 **  
    (_targetToken.level);  
  
    emit AssignNumber({  
        tokenNumber: tokenNumber,  
        targetNumber: targetNumber,  
        backingAmount: 2 ** (_targetToken.level),  
        totalAmount: assignedNumbers[targetNumber]  
    });  
}
```

Depending how this number is used, let's consider the following scenario:

- 1) Alice purchased multiple level 19 tokenIDs and assigned the number 1 to all these tokens, hence the number 1 will have a large value. All her tokens point towards this desired number.
- 2) Bob purchases a level 10 token and assigns the number 1 towards this token, this will increase the overall value assigned to number 1, but Bob now also has one token assigned to this number.

Depending on the downstream use-case, Bob might be exposed to the benefits for number 1, since he also has a token assigned to that number. Since we currently do not know the specific idea behind this, this should be kept in mind.

Recommendation Consider the exact downstream use-case of the assigned number value.

An ideal scenario would be to only allow users to assign the number of a tokenId they in fact own, that way they can represent unique ownership towards that number, however, that would require further auditing techniques to ensure there will not occur any issues when redeeming or burning.

Maybe this specific number is also inherently connected towards a specific user address in another contract — this would also be a way which makes sense, since users then would only allocate tokenIds to their own private number.

Finally, it comes down on how the uniqueness and ownership of the desired number/license number is secured.

Resolution

 ACKNOWLEDGED

The client mentioned that this is desired.

Issue #07 **String is not backed by values**

Severity

 LOW SEVERITY

Description

While the assignNumber function allows the token to have a specific number assigned, the assignString function also allows the token to have a specific string assigned.

The only issue is that the assigned number will store the aggregate of all tokenId values, while the assigned string will not. Therefore, assigning a string towards a tokenId seems likely redundant.

Recommendation Consider if it is desired to also accumulate aggregated values towards a specific string. If yes, consider implementing this feature.

Resolution

 RESOLVED

It is now possible to back a string with values.

Issue #08 Redeeming an NFT can make a level exceed the LEVEL_LIMIT number of NFT	
Severity	INFORMATIONAL
Description	Whenever users decide to redeem their tokenId, this tokenId will then be pushed into the level, without worrying about the level limit.
Recommendation	This is only considered as an informational issue only, as we assume that this behavior is expected and should not be changed, otherwise this might prevent users from redeeming their NFTs.
Resolution	ACKNOWLEDGED The client mentioned that this is desired.

Issue #09 Token split can be DoS'ed	
Severity	INFORMATIONAL
Description	A token split can be simply DoS'ed by another user frontrunning the function or redeeming a token such that the gap in the destination level is insufficient.
Recommendation	This is only considered as an informational issue as we assume that users would split tokens to then buy them, so in this case the user would simply buy the tokens newly added to the level.
Resolution	ACKNOWLEDGED

Issue #10	The tree is hard to query
Severity	 INFORMATIONAL
Description	Users can use the levels function to query NFTs that are in the level one by one, but they can't know the number of nfts in each level. This will make it very hard to query the entire tree as most of the levels would revert because of an out of length error.
Recommendation	Consider adding a length function for each level of the tree.
Resolution	 RESOLVED



Description

L215

```
if (!_exists(tokenNumber) || nftTokens[tokenNumber].minted == false)
```

Consider updating the second check to ! nftTokens[tokenNumber].minted) save some gas and make the code easier to read.

L224

```
if (nftTokens[tokenNumber].minted == false) {
```

Similarly, this check should be updated to if (! nftTokens[tokenNumber].minted)

L257-273

```
function addRootTokens(uint256 tokens) external onlyOwner {  
    //@dev _rootData.endRootNumber is inclusive, so we need  
    to add 1 to it for this check.  
    if (_rootData.currentRootNumber + tokens >  
        _rootData.endRootNumber + 1) {        revert  
        TooManyRootTokens();  
    }  
  
    for (uint256 i; i < tokens; i++) {  
        levels[MAX_LEVEL].push(_rootData.currentRootNumber +  
i);  
        nftTokens[_rootData.currentRootNumber + i] =  
TokenData('', 0, MAX_LEVEL, false);  
        emit AddRootToken({  
            rootNumber: _rootData.currentRootNumber + i  
        });  
    }  
  
    _rootData.currentRootNumber += tokens;  
    _updateNFTArray(MAX_LEVEL);  
}
```

Consider caching currentRootNumber to save some gas.

L371

```
this.safeTransferFrom(address(this), msg.sender,  
tokenNumber);
```

Consider using the internal function `_safeTransfer` to save some gas. This will still check that `address(this)` is the owner of the token `tokenNumber`.

L457

```
this.assignString(tokenNumber, data);
```

This should use the internal function `_assignString` to save some gas.

L502

```
return this.getAssignmentString(tokenNumber);
```

This function should return `nftTokens[tokenNumber].assignedString`; directly to save some gas. The logic can also be moved to an internal function and used in `getAssignmentString` and `getBurnData` for cleaner code.

Recommendation	Consider implementing the gas optimizations mentioned above.
-----------------------	--

Resolution	 RESOLVED
-------------------	--



PALADIN
BLOCKCHAIN SECURITY