Hypercerts Security Review

Pashov Audit Group

Conducted by: pashov

February 17th, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Hypercerts** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Hypercerts

Hypercerts is a protocol that allows for Impact Funding Systems (IFSs) to be built efficiently on the blockchain. Users can mint ERC1155 semi-fungible tokens that are like "certificates" for work. Those "certificates" can be configured to be fractionalized and/or transferable based on rules. One great use case for Hypercerts is the retrospective funding mechanism since once you have the "certificate" you will be able to get the retrospective funding payments coming in the future. Minted Hypercerts can be split (fractionalized) and merged. Fractionalization is a helpful feature when you want to give/sell a part of your expected rewards to another party.

There are 3 ways for anyone to create a new Hypercert (also called a "claim"):

1. Calling `mintClaim` which will mint all of the `units` of the token to the caller
2. Calling `mintClaimWithFractions` which will split the token to `fractions` and mint them to the caller
3. Calling `createAllowlist` which will create an ERC1155 token that will need a whitelist access (via a Merkle tree) for a caller to mint it

For the functionality in 3. the following methods for minting a new Hypercert were added:

1. `mintClaimFromAllowlist` - the caller can mint to `account` by submitting a `proof` which authorizes him to mint `units` amount of the `claimID` type token
2. `batchMintClaimsFromAllowlists` - same as `mintClaimFromAllowlist` but for multiple mints in a single transaction

And for the owners of Hypercerts/claims the following functionalities exist:

1. `splitValue` - split a claim token into fractions
2. `mergeValue` - merge fractions into one claim token
3. `burnValue` - burn claim token

Threat Model

System Actors

- Protocol owner - can upgrade the `HypercertMinter` contract and pause/unpause it, set during protocol initialization
- Claim minter - can create a new claim type, no authorization control
- Whitelisted minter of a claim - can mint tokens from an already existing type, Merkle tree root is set during creation of the type
- Type creator - if policy is `FromCreatorOnly` only him can transfer the tokens
- Fraction owner - can transfer, burn, split and merge fractions of a claim

Q: What in the protocol has value in the market?

A: The claims and their fractional ownership are valuable because they might receive rewards in the future from (for example) retroactive funding and/or be used for governance.

Q: What is the worst thing that can happen to the protocol?

1. Stealing/burning claims by a user who doesn't own and isn't an operator of the claim
2. Generating more units than intended via splitting or merging
3. Unauthorized upgrading/pausing of the contract

Interesting/unexpected design choices:

The `mintClaimFromAllowlist` method checks `msg.sender` to be included in the Merkle tree but the token is minted to the `account` address argument instead. The same is the case for the `batchMintClaimsFromAllowlists` functionality where `msg.sender` should be in all of the leafs.

Minting a token with only 1 unit means it won't be splittable at a later stage. The UI recommends 100 fractions on mint - maybe this should be enforced on the smart contract level as a minimum value.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

***review commit hash* - 73cd850e96d4fd50924640b838bcc0f6905b0abf**

Scope

The following smart contracts were in scope of the audit:

- `AllowlistMinter`
- `HypercertMinter`
- `SemiFungible1155`
- `Upgradeable1155`
- `interfaces/**`
- `libs/**`

7. Executive Summary

Over the course of the security review, pashov engaged with Hypercerts to review Hypercerts. In this period of time a total of **20** issues were uncovered.

Protocol Summary

Protocol Name	Hypercerts
Date	February 17th, 2023

Findings Count

Severity	**Amount**
Critical	1
High	1
Medium	2
Low	4
QA	12
Total Findings	**20**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Users can split a token to more fractions than the units held at tokenID	Critical	Resolved
[H-01]	Calling splitValue when token index is not the latest will overwrite other claims' storage	High	Resolved
[M-01]	Unused function parameters can lead to false assumptions on user side	Medium	Resolved
[M-02]	Input & data validation is missing or incomplete	Medium	Resolved
[L-01]	Comment has incorrect and possible dangerous assumptions	Low	Resolved
[L-02]	Missing event and incorrect event argument	Low	Resolved
[L-03]	Prefer two-step pattern for role transfers	Low	Resolved
[L-04]	Contracts pausability and upgradeability should be behind multi-sig or governance account	Low	Resolved
[QA-01]	Transfer hook is not needed in current code	QA	Resolved
[QA-02]	Unused import, local variable and custom errors	QA	Resolved
[QA-03]	Merge logic into one smart contract instead of using inheritance	QA	Resolved
[QA-04]	Incorrect custom error thrown	QA	Resolved
[QA-05]	Typos in comments	QA	Resolved
[QA-06]	Missing override keyword for interface inherited methods	QA	Resolved

[QA-07]	Bit-shift operations are unnecessarily complex	QA	Resolved
[QA-08]	Redundant check in code	QA	Resolved
[QA-09]	Incomplete or wrong NatSpec docs	QA	Resolved
[QA-10]	Misleading variable name	QA	Resolved
[QA-11]	Solidity safe pragma best practices are not used	QA	Resolved
[QA-12]	Magic numbers in the codebase	QA	Resolved

8. Findings

8.1. Critical Findings

[C-01] Users can split a token to more fractions than the `units` held at `tokenID`

Severity

Impact

High, as it breaks an important protocol invariant

Likelihood

High, as those types of issues are common and are easily exploitable

Description

The `_splitValue` method in `SemiFungible1155` does not follow the Checks-Effects-Interactions pattern and it calls `_mintBatch` from the ERC1155 implementation of OpenZeppelin which will actually do a hook call to the recipient account as a safety check. This call is unsafe as it can reenter the `_splitValue` method and since `tokenValues[_tokenID]` hasn't been updated yet, it can once again split the tokens into more fractions and then repeat until a huge amount of tokens get minted.

Recommendation

Follow the Checks-Effects-Interactions pattern

```diff
-_mintBatch(_account, toIDs, amounts, "");
-
-tokenValues[_tokenID] = valueLeft;
+tokenValues[_tokenID] = valueLeft;
+
+_mintBatch(_account, toIDs, amounts, "");
```

Discussion

pashov: Client has fixed the issue.

8.2. High Findings

[H-01] Calling `splitValue` when token index is not the latest will overwrite other claims' storage

Severity

Impact: High, as it can lead to loss of units for an account without any action on his side

Likelihood: Medium, because it can happen only with a token that has a non-latest index

Description

The logic in `_splitValue` is flawed here:

```
uint256 currentID = _tokenID;
...
toIDs[i] = ++currentID;
...
for (uint256 i; i < len; ) {
    valueLeft -= values[i];

    tokenValues[toIDs[i]] = values[i];

    unchecked {
        ++i;
    }
}
...
_mintBatch(_account, toIDs, amounts, "");
```

Let's look at the following scenario:

1. Alice mints through allowlist, token 1, 10 units
2. Bob mints through allowlist, token 2, 100 units
3. Alice calls `splitValue` for token 1 to 2 new tokens, both 5 units

Now we will have `tokenValues[toIDs[i]] = values[i]` where `toIDs[i]` is `++currentID` which is 2 and `values[i]` is 5, so now `tokenValues[2] = 5` which is overwriting the `tokenValues` of Bob. Also, later `_mintBatch` is called with Bob's token ID as a token ID, which will make some of the split tokens be of the type of Bob's token.

Recommendations

Change the code the following way:

```
- maxIndex[_typeID] += len;
...
- toIDs[i] = ++currentID;
+ toIDs[i] = _typeID + ++maxIndex[typeID];
```

Discussion

pashov: Client has fixed the issue.

8.3. Medium Findings

[M-01] Unused function parameters can lead to false assumptions on user side

Severity

Impact: Low, because the caller still controls the minted token

Likelihood: High, because users will provide values to those parameters and their assumptions about their usage will always be false

Description

The `units` parameter in `mintClaimWithFractions` is used only in the event emission. This is misleading as actually `fractions.length` number of fractions will be minted. If `units != fractions.length` this can have unexpected consequences for a user. The same is the problem with the `account` parameter in both `mintClaim` and `mintClaimWithFractions` - it is not used in the method and actually `msg.sender` is the account to which tokens are minted and is set as the token creator. Again if `account != msg.sender` this is unexpected from a user standpoint and while in the best case scenario leads to a not so great UX, in the worst case it can lead to faulty assumptions for value received by the `account` address.

Recommendations

Remove the `units` parameter from `mintClaimWithFractions` and also use `account` instead of `msg.sender` in the `_mintValue` call in `mintClaim` and `mintClaimWithFractions`.

Discussion

pashov: Client has fixed the issue.

[M-02] Input & data validation is missing or incomplete

Severity

Impact: High, because in some cases this can lead to DoS and unexpected behaviour

Likelihood: Low, as it requires malicious user or a big error on the user side

Description

Multiple methods are missing input/data validation or it is incomplete.

1. The `splitValue` method in `SemiFungible1155` has the `maxIndex[_typeID] += len;` code so should also do a `notMaxItem` check for the index
2. The `createAllowlist` method accepts a `units` argument which should be the maximum units mintable through the allowlist - this should be enforced with a check on minting claims from allowlist
3. The `_createAllowlist` of `AllowlistMinter` should revert when `merkleRoot == ""`
4. The `_mintClaim` and `_batchMintClaims` methods in `SemiFungible1155` should revert when `_units == 0` or `_units[i] == 0` respectively

Discussion

pashov: Client has partially fixed the issue.

Recommendations

Add the checks mentioned for all inputs and logic.

8.4. Low Findings

[L-01] Comment has incorrect and possible dangerous assumptions

The comment in the end of `SemiFungible1155` assumes constant values are saved in the contract storage which is not the case. Both constants and immutable values are not stored in contract storage. Update comment and make sure to understand the workings of smart contracts storage so it does not lead to problems in upgrading the contracts in a later stage.

Discussion

pashov: Client has fixed the issue.

[L-02] Missing event and incorrect event argument

The `_mergeValue` method in `SemiFungible1155` does not emit an event while `_splitValue` does - consider emitting one for off-chain monitoring. Also the `TransferSingle` event emission in `_createTokenType` has a value of 1 for the `amount` argument but it does not actually transfer or mint a token so the value should be 0.

Discussion

pashov: Client has fixed the issue.

[L-03] Prefer two-step pattern for role transfers

The `Upgradeable1155` contract inherits from `OwnableUpgradeable` which uses a single-step pattern for ownership transfer. It is a best practice to use two-step ownership transfer pattern, meaning ownership transfer gets to a "pending" state and the new owner should claim his new rights, otherwise the old owner still has control of the contract. Consider using a two-step approach.

Discussion

pashov: Client has acknowledged the issue.

[L-04] Contracts pausability and upgradeability should be behind multi-sig or governance account

A compromised or a malicious owner can call `pause` and then `renounceOwnership` to execute a DoS attack on the protocol based on pausability. The problem has an even wider attack surface with upgradeability - the owner can upgrade the contracts with arbitrary code at any time. I suggest using a multi-sig or governance as the protocol owner after the contracts have been live for a while or using a Timelock smart contract.

Discussion

pashov: Client is taking measures to move into this direction.

8.5. QA Findings

[QA-01] Transfer hook is not needed in current code

The `_beforeTokenTransfer` hook in `SemiFungible1155` is not needed as it only checks if a base type token is getting transferred but the system in its current state does not actually ever mint such a token, so this check is not needed and only wastes gas. Remove the `_beforeTokenTransfer` hook override in `SemiFungible1155`.

Discussion

pashov: Client has fixed the issue.

[QA-02] Unused import, local variable and custom errors

The `IERC1155ReceiverUpgradeable` import in `SemiFungible1155` is not actually used and should be removed, same for the `_typeID` local variable in `SemiFungible1155::mergeValue`, same for the `ToZeroAddress`, `MaxValue` and `FractionalBurn` custom errors.

Discussion

pashov: Client has fixed the issue.

[QA-03] Merge logic into one smart contract instead of using inheritance

The `SemiFungible1155` contract inherits from `Upgradeable1155` but it doesn't make sense to separate those two since the logic is very coupled and `Upgradeable1155` won't be inherited from other contracts so it does not need its own abstraction. Merge the two contracts into one and give it a good name as the currently used two names are too close to the `ERC1155` standard.

Discussion

pashov: Client has fixed the issue.

[QA-04] Incorrect custom error thrown

The code in `AllowlistMinter::_processClaim` throws `DuplicateEntry` when a leaf has been claimed already - throw an `AlreadyClaimed` custom error as well. Also consider renaming the `node` local variable there to `leaf`.

Discussion

pashov: Client has fixed the issue.

[QA-05] Typos in comments

`AlloslistMinter` -> `AllowlistMinter`

Discussion

pashov: Client has fixed the issue.

[QA-06] Missing `override` keyword for interface inherited methods

The `HypercertMinter` contract is inheriting the `IHypercertToken` interface and implements its methods but the `override` keyword is missing on the overriden methods. Add the keyword on those as a best practice and for compiler checks.

Discussion

pashov: Client has fixed the issue.

[QA-07] Bit-shift operations are unnecessarily complex

I recommend the following change for simplicity:

```diff
-    uint256 internal constant TYPE_MASK = uint256(uint128(int128(~0))) << 128;
+    uint256 internal constant TYPE_MASK = type(uint256).max << 128;

    /// @dev Bitmask used to expose only lower 128 bits of uint256
-    uint256 internal constant NF_INDEX_MASK = uint128(int128(~0));
+    uint256 internal constant NF_INDEX_MASK = type(uint256).max >> 128;
```

Discussion

pashov: Client has fixed the issue.

[QA-08] Redundant check in code

The `_beforeValueTransfer` hook in `SemiFungible1155` has the `getBaseType(_to) > 0` check which always evaluates to `true` so it can be removed.

Discussion

pashov: Client has fixed the issue.

[QA-09] Incomplete or wrong NatSpec docs

The NatSpec docs on the external methods are incomplete - missing `@param`, `@return` and other descriptive documentation about the protocol functionality. Also part of the NatSpec of `IAllowlist` is copy-pasted from `IHypercertToken`, same for `AllowlistMinter`. Make sure to write descriptive docs for each method and contract which will help users, developers and auditors.

Discussion

pashov: Client still hasn't fixed the issue but is setting higher priority to document the code better.

[QA-10] Misleading variable name

In the methods `mintClaim`, `mintClaimWithFractions` and `createAllowlist` the local variable `claimID` has a misleading name as it actually holds `typeID` value - rename it to `typeID` in the three methods.

Discussion

pashov: Client has fixed the issue.

[QA-11] Solidity safe pragma best practices are not used

Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. The project is currently using a floatable version.

Discussion

pashov: Client has fixed the issue.

[QA-12] Magic numbers in the codebase

In `SemiFungible1155` we have the following code:

```
if (_values.length > 253 || _values.length < 2) revert Errors.ArraySize();
```

Extract the `253` value to a well-named constant so the intention of the number in the code is clear.

Discussion

pashov: Client has fixed the issue.