SECURITY REVIEW OF FANTASY

Summary

Auditors: 0xWeiss (Marc Weiss)

Client: Fantasy

Report Delivered: 21 April, 2024

About 0xWeiss

0xWeiss is an independent security researcher. In-house auditor/security engineer in Ambit Finance and Tapioca DAO. Security Researcher at Paladin Blockchain Security and ASR at Spearbit DAO. Reach out on Twitter @0xWeisss .

Protocol Summary

Fantasy is a Trading Card Game in which players collect cards featuring crypto influencers to compete and earn ETH, BLAST, more cards, and FAN Points.

Protocol Name	Fantasy
Language	Solidity
Codebase	https://github.com/fantasy-top/fantasy-core-audit
Commit	4cc424eb49f036c94656dd1c916be4cf891a5c1b
Previous Audits	Cantina

Audit Summary

Fantasy engaged **0xWeiss** through Hyacinth to review the security of its codebase.

A 2 week time-boxed security assesment was performed.

At the end, 9 issues were identified.

All findings have been recorded in the following report. Notice that the examined smart contracts are not resistant to internal exploit.

For a detailed understanding of risk severity, source code vulnerability, and potential attack vectors, refer to the complete audit report below.

Vulnerability Summary

Severity	Total	Pending	Acknowledged	Par. resolved	Resolved
🔴 HIGH	**0**	**0**	**0**	**0**	**0**
🟡 MEDIUM	**3**	**0**	**2**	**0**	**1**
🟢 LOW	**6**	**0**	**5**	**0**	**1**
🔵 INF	**0**	**0**	**0**	**0**	**0**

Severity Classification

Severity	Classification
🔴 HIGH	Exploitable, causing loss/manipulation of assets or data.
🟡 MEDIUM	Risk of future exploits that may or may not impact the smart contract execution.
🟢 LOW	Minor code errors that may or may not impact the smart contract execution.
🔵 INF	No impact issues. Code improvement

Methodology

The auditing process pays special attention to the following considerations:

● Testing the smart contracts against both common and uncommon attack vectors.

● Assessing the codebase to ensure compliance with current best practices and industry standards.

● Ensuring contract logic meets the specifications and intentions of the client.

● Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.

● Thorough line-by-line manual review of the entire codebase by industry experts.

Audit Scope

ID	File Path
ED	src/ExecutionDelegate.sol
FC	src/FantasyCards.sol
MINT	src/Minter.sol
OL	src/libraries/OrderLib.sol
VRG	src/VRGDA/VRGDA.sol
LVRG	src/VRGDA/LinearVRGDA.sol
WM	src/VRGDA/wadMath.sol
EXC	src/Exchange.sol

Findings and Resolutions

ID	Category	Severity	Status
EXC-M1	User Loss	🟡 **MEDIUM**	Resolved
EXC -M2	User Loss	🟡 **MEDIUM**	Acknowledged
EXC -M3	User Loss	🟡 **MEDIUM**	Acknowledged
MINT-L1	Logical error	🟢 **LOW**	Acknowledged
MINT-L2	Logical error	🟢 **LOW**	Acknowledged
MINT-L3	Logical error	🟢 **LOW**	Acknowledged
MINT-L4	DOS	🟢 **LOW**	Acknowledged
EXEC-L1	Input Validation	🟢 **LOW**	Resolved
GLOBAL-L2	Un-used code	🟢 **LOW**	Acknowledged

[EXC-M1] Orders can be executed on expiration

Severity	Category	Status
🟡 MEDIUM	User Loss	Resolved

Description

When buying or selling Fantasy cards, there is the following check that requires that the order is not expired:

```
require(buyOrder.expirationTime >= block.timestamp, "order expired");
```

The problem is that usually when speaking about any type of orders in the defi space, when an order reaches its expiration time, the order is already expired. Therefore, you should not be able to execute `buy()` or `sell()` orders when the `expirationTime` has just been reached.

Recommendation

Update the require statements so that it does not allow to execute buy and sell orders just at expiration:

```
function _buy(OrderLib.Order calldata sellOrder, bytes calldata sellerSign
ature) internal {
       require(sellOrder.side == OrderLib.Side.Sell, "order must be a sell
");
-        require(sellOrder.expirationTime >= block.timestamp, "order expir
ed");
+        require(sellOrder.expirationTime > block.timestamp, "order expire
d");
       require(sellOrder.trader != address(0), "order trader is 0");

function sell(OrderLib.Order calldata buyOrder,bytes calldata buyerSignatu
re,uint256 tokenId,bytes32[] calldata merkleProof) public payable nonReent
rant onlyEOA {
       require(buyOrder.paymentToken != address(0), "payment token can not
be ETH for buy order");
       require(buyOrder.side == OrderLib.Side.Buy, "order must be a buy");
-        require(buyOrder.expirationTime >= block.timestamp, "order expire
d");
+        require(buyOrder.expirationTime > block.timestamp, "order expired
");
```

Resolution

Fixed

[EXC-M2] Malicious seller can grief order executions

Severity	Category	Status
● MEDIUM	User Loss	Acknowledged

Description

Currently, on the `Exchange.sol` contract, a seller signs their message with the corresponding data that the buyer will use to execute the `buy()` order:

```
 function buy(
        OrderLib.Order calldata sellOrder,
        bytes calldata sellerSignature
    ) public payable nonReentrant onlyEOA {
        _buy(sellOrder, sellerSignature);
    }
```

Then, it checks that the actual seller specified, is the same seller that signed the message: `require(sellOrderSigner == sellOrder.trader, "invalid signature");`

Finally, the Fantasy card is transferred from the seller to the buyer through `_executeTokenTransfer(sellOrder.collection, sellOrder.trader, msg.sender, sellOrder.tokenId);`

Here is where the griefing vector comes into play. A malicious seller that just wants to grief buyers, would just create orders and front-run buyers that want to buy their fantasy card and `revoke approval of their` NFT through `ExecutionDelegate.sol`:

```
 function transferERC721Unsafe(
        address collection,
        address from,
        address to,
        uint256 tokenId
    ) external whenNotPaused approvedContract {
        require(revokedApproval[from] == false, "User has revoked ap
proval");
        IERC721(collection).transferFrom(from, to, tokenId);
    }
```

Therefore, the malicious seller would just call `revokeApproval()` front-running the `buy()` call of the buyer from the `Exchange`.

Recommendation

Re-engineer this mechanism to account for griefings, probably a system where you can’t revoke after an order is live.

Resolution

Acknowledged

[EXC-M3] Rarities can’t be granted on the smart contract level, Possibly losing prestigious fantasy cards while leveling up.

Severity	Category	Status
🟡 MEDIUM	User Loss	Acknowledged

Description

Currently, on the `levelUp()` function, users should `upgrade their hero card to the next level of rarity by burning a specified number of cards of the same hero and rarity.` The problem is that the `levelUp()` function does not grant that functionality allowing for users that interact with the smart contracts directly, not mint the correct rarity of card and lose their hero.

```
 function levelUp(uint256[] calldata tokenIds, address collection) p
ublic {
       require(tokenIds.length == cardsRequiredForLevelUp, "wrong am
ount of cards to level up");

       for (uint i = 0; i < cardsRequiredForLevelUp; i++) {
           require(
               IFantasyCards(collection).ownerOf(tokenIds[i]) == msg
.sender,
               "caller does not own one of the tokens"
           );
           executionDelegate.burnFantasyCard(address(collection), to
kenIds[i]);
       }

       uint256 mintedTokenId = IFantasyCards(collection).tokenCounte
r();
       executionDelegate.mintFantasyCard(address(collection), msg.se
nder);
       emit LevelUp(tokenIds, mintedTokenId, collection, msg.sender)
;
   }
```

Recommendation

Adopt rarities on the smart contract level so that it does not have to be handled on the front-end.

Resolution

Acknowledged. This issue is fixed at the front-end where the team will make sure to set the right metadata for specific leveled up cards, though on the contracts, there is no way to handle the rarity functionality.

[MINT-L1] setMaxPacksForMintConfig can be set below the current minted packs

Severity	Category	Status
● LOW	Logical error	Acknowledged

Description

On the `setMaxPacksForMintConfig` it allows the MASTER to set the max number of packs that can be minted for that `mintConfigId`.

This can be called and updated anytime, and should follow the INVARIANT that specifies that `maxPacks` that can be minted, has to be bigger or equal to the current amount of packs minted `totalMintedPacks`:

`config.maxPacks >= config.totalMintedPacks`

```
  function setMaxPacksForMintConfig(uint256 mintConfigId, uint256 maxPacks) public onlyRole(MINT_CONFIG_MASTER) {
        require(mintConfigId < mintConfigIdCounter, "Invalid mintConfigId");
        require(maxPacks > 0, "Maximum packs must be greater than 0");
        MintConfig storage config = mintConfigs[mintConfigId];
        config.maxPacks = maxPacks;

        emit MaxPacksUpdatedForMintConfig(mintConfigId, maxPacks);
    }
```

Though there is no check to prevent an incorrect state by setting `maxPacks` below `totalMintedPacks`. # Recommendation

Add the following code:

```
+ if (maxPacks > config.totalMintedPacks){
+    config.maxPacks = config.totalMintedPacks;
+ }else{
+ config.maxPacks = maxPacks;
+ }
- config.maxPacks = maxPacks;
```

Resolution

Acknowledged

[MINT-L2] getPackPrice() should revert if the `mintConfig` has been canceled

Severity	Category	Status
● LOW	Logical error	Acknowledged

Description

Currently, when calling `getPackPrice()` externally, you will receive incorrect data/states from reality as you can get prices for canceled `configIds`:

Recommendation

Add a requirement so that if the `configIds` has been canceled, revert:

```
VRGDAConfig memory vrgdaConfig = mintConfig.vrgdaConfig;
require((block.timestamp - mintConfig.startTimestamp) >= 0, "INVALID_TIMES
TAMP");
+ require(!mintConfig.cancelled, "CANCELLED ID");
```

Resolution

Acknowledged

[MINT-L3] getPackPrice() should revert if the `mintConfig` has an expired timestamp

Severity	Category	Status
● LOW	Logical error	Acknowledged

Description

Currently, when calling `getPackPrice()` externally, you will receive incorrect data/states from reality as you can get prices for expired `configIds`.

Recommendation

Add a requirement so that if the `configIds` has been expired, revert:

```
VRGDAConfig memory vrgdaConfig = mintConfig.vrgdaConfig;
require((block.timestamp - mintConfig.startTimestamp) >= 0, "INVALID_TIMES
TAMP");
+ require(mintConfig.expirationTimestamp > block.timestamp, "INVALID_TIMES
TAMP");
```

Resolution

Acknowledged

[MINT-L4] Lack of upper limit in `cardsPerPack` could cause a DOS when minting

Severity	Category	Status
● LOW	DOS	Acknowledged

Description

When calling `mint()` on the Minter contract, you are in fact buying one pack of cards of whatever collection is specified in the `configId` you specified.

At the end of the function, you will start batch minting the cards until all the cards on the pack have been minted:

```
    function _executeBatchMint(address collection, uint256 cardsPerP
ack, address buyer) internal {
        for (uint256 i = 0; i < cardsPerPack; i++) {
            executionDelegate.mintFantasyCard(collection, buyer);
        }
    }
```

If the amount of cards in the pack is high enough, the transaction will reach the gas limit and revert, not allowing to mint the specified pack of fantasy cards.

Recommendation

Add an upper limit when setting `cardsPerPack` to a reasonable value, I estimate around 50:

```
+ uint256 cardLimit;

  function setCardsPerPackForMintConfig(
        uint256 mintConfigId,
        uint256 cardsPerPack
    ) public onlyRole(MINT_CONFIG_MASTER) {
        require(mintConfigId < mintConfigIdCounter, "Invalid mintConfigId"
);
        require(cardsPerPack > 0, "Cards per pack must be greater than 0")
;
        MintConfig storage config = mintConfigs[mintConfigId];
+       require(cardsPerPack <= cardLimit, "too many cards per pack");
        config.cardsPerPack = cardsPerPack;
        emit CardsPerPackUpdatedForMintConfig(mintConfigId, cardsPerPack);
    }
```

```
 function newMintConfig(
        address collection,
        uint256 cardsPerPack,
        uint256 maxPacks,
        address paymentToken,
        uint256 fixedPrice,
        uint256 maxPacksPerAddress,
        bool requiresWhitelist,
        bytes32 merkleRoot,
        uint256 startTimestamp,
        uint256 expirationTimestamp
    ) public onlyRole(MINT_CONFIG_MASTER) {
        require(collection != address(0), "Collection address cannot be 0x
0");
        require(cardsPerPack > 0, "Cards per pack must be greater than 0")
;
        require(maxPacks > 0, "Max packs must be greater than 0");
        require(startTimestamp >= block.timestamp, "Mint must start immedi
ately or in the future");
        require(expirationTimestamp == 0 || expirationTimestamp > startTim
estamp, "invalid expirationTimestamp");
+       require(cardsPerPack <= cardLimit, "too many cards per pack");

        if (requiresWhitelist) {
            require(merkleRoot != 0, "missing merkleRoot");
        }

        MintConfig storage config = mintConfigs[mintConfigIdCounter];
        config.collection = collection;
        config.cardsPerPack = cardsPerPack;
```

Resolution

Acknowledged

[EXEC-L1] “Fake” volume and “fake” mintings can happen

Severity	Category	Status
● LOW	Input Validation	Resolved

Description

Functions like `mintFantasyCard` on `ExecutionDelegate.sol` allow the Minter contract mint NFTs form a `IFantasyCards` contract, usually specified inside the `mintConfigs.collection` parameter:

```
 function mintFantasyCard(address collection, address to) external w
henNotPaused approvedContract {
        IFantasyCards(collection).safeMint(to);
    }
```

Though, for functions like `levelUp` and `burnToDraw()`, the caller is allowed to to specify whatever `collection` they want with no specific validation or whatsoever.

```
 function levelUp(uint256[] calldata tokenIds, address collection) p
ublic {
        require(tokenIds.length == cardsRequiredForLevelUp, "wrong a
mount of cards to level up");

        for (uint i = 0; i < cardsRequiredForLevelUp; i++) {
            require(
                IFantasyCards(collection).ownerOf(tokenIds[i]) == ms
g.sender,
                "caller does not own one of the tokens"
            );
            executionDelegate.burnFantasyCard(address(collection), t
okenIds[i]);
        }

        uint256 mintedTokenId = IFantasyCards(collection).tokenCount
er();
        executionDelegate.mintFantasyCard(address(collection), msg.s
ender);
        emit LevelUp(tokenIds, mintedTokenId, collection, msg.sender
);
    }
```

allowing for phantom contracts with the `IFantasyCards` to be used as real Fantasy Cards. Impact is not more than faking real volumen with fake Fantasy Card collections and no funds will be lost.

Recommendation

Add a whitelist system for all the collections that will be added to every `mintConfigs.collection` , and check against this whitelist when calling `mintFantasyCard()` and `burnFantasyCard()`

Resolution

Fixed

[GLOBAL-L1] Un-used imports across the codebase

Severity	Category	Status
● LOW	Un-used code	Acknowledged

Description

Remove the following imports from their contracts as they are declared but not used.

- `IFantasyCards.sol` import: `import "@openzeppelin/contracts/interfaces/draft-IERC6093.sol";` is un-used.

Resolution

Acknowledged

DISCLAIMER

Most of the Acknowledged issues on this report are acknowledged because the team had no time to fix given an extremely tight deadline for deployment. That is also the reason why the review only was of 2 days of duration.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Marc Weiss to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Blockchain technology and cryptographic assets present a high level of ongoing risk.

My position is that each company and individual are responsible for their own due diligence and continuous security. My goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze. Therefore, I do not guarantee the explicit security of the audited smart contract, regardless of the verdict.