PxTrainerAdventure audit report

Scope of Work

The scope of this audit is based on the material provided by the client. The goal of this audit is to ensure the following:

- Find potential exploits for the contract
- Find issues in the contract that will affect the integrity of the mint and claim processes
- Ensure the contract adheres to the business logics provided by the client

The logic of the contract will be compared to the business logic contained in the materials provided below:

- **3 contracts** were provided in total
  - PxChainlinkManager
  - PxTrainerAdventure
  - PxWeekManager
- **Simple contracts logic description**

The contracts code was provided on 24th March 2023. Audit report was delivered on 28th March 2023. It was given in the following formats:

- **PDF document**
- **zip folder with 2 .sol attachments as a reference for the gas optimization changes**

Disclaimer

This is a limited report based on our analysis of the Smart Contract audit and performed in accordance with best practices as of the date of this report, in relation to cybersecurity vulnerabilities and issues in the smart contract source code analyzed, the details of which are set out in this report, (hereinafter “Source Code”) and the Source Code compiling, deploying and performing the intended functions. In order to get a full view of our findings and the scope of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn’t say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions.

Summary

The contracts are well-documented, but sometimes they lack a good logical division. A few variables and methods working with these variables are divided into separate contracts. Some of these moments were not mentioned in this report since they do not affect the workflow. However, they may make the analysis and future contract maintenance more challenging. Some of them were mentioned as low-priority problems, as they affect gas costs.

Several problems mentioned in the report directly affect the workflow and make it not possible to work correctly. These issues were partially found during the edge cases tests. Therefore, I would recommend conducting tests that will cover all of these moments, including:

- Claiming the treasure pool fully and refilling it
- Claiming treasures with an available amount of 1
- Moving to the next week
- etc.

Please ensure these tests are completed prior to the launch.

No problems were found in **PxWeekManager** contract, so this section was omitted in report.

Findings summary

	Critical	Medium	Low
PxChainlinkManager	0	0	1
PxTrainerAdventure	1	1	0
PxWeekManager	0	0	0
Total	1	1	1

PxChainlinkManager

0 Critical
0 Medium
1 Low

Low **Functions that called from other contract only**

Description:

`setSignerAddress` and `isSignerVerifiedFromSignature` functions are only called on the `PxTrainerAdventure` contract side and most likely should be placed within it's codebase.

Impact:

Calling a remote contracts functions will always require more gas than if they were placed directly to the caller's code. In case of `isSignerVerifiedFromSignature` - it's called during the claim process, which means that every user that claim tokens will pay an additional gas. Refer to **Gas optimizations** part in the very end of this report for more details.

How to fix:

Moving both these functions to `PxTrainerAdventure` will reduce the gas costs for the `claimTreasure` function.

PxTrainerAdventure

1 Critical
1 Medium
0 Low

Critical **Logical flaws in ERC721 transfer token function check**

Description:

Transfer **ERC721** token function will throw an error in case if amount of specific Treasure tokens claimed is equal to the total amount of this Treasure's tokens.

```
function transferToken(uint256 _weekNumber, Treasure memory _treasure) internal {
...
if (_treasure.tokenIds.length == _treasure.claimedToken) {
                revert InsufficientToken();
}
```

At the same time `_treasure.claimedToken` variable will be increased prior to transfer for every claim.

Impact:

Every time the last token of ERC721 `Treasure` will be claimed - the tranfer will fail. It will also fail if only one token of collection was added as `Treasure`.

How to fix:

```
if (_treasure.tokenIds.length < _treasure.claimedToken) {
                revert InsufficientToken();
}
```

Medium **TreasureTransferred Event does not contain the data required**

Description:

```
/// @notice Struct object to store treasure information
/// @dev If the treasure is ERC1155,tokenIds is an  empty array
///      if the treasure is ERC721,tokenId value is a dummy
struct Treasure {
        address collectionAddress;
        uint256 tokenId;
        uint256[] tokenIds;
        uint256 claimedToken;
        uint8 contractType;
        uint8 treasureType;
}

event TreasureTransferred(uint256 weekNumber, address userWallet, address collectionAddress, uint256
tokenId, uint256 tokenType);

emit TreasureTransferred(_weekNumber, msg.sender, _treasure.collectionAddress, _treasure.tokenId,
_treasure.contractType);
```

Impact:

Whenever the **ERC721** token is transfered the Event with tokenId 0 will be emitted, impact depends on what backend logic depends on these events.

How to fix:

The easies way to fix it is to store the real ID of transferred token (`_treasure.tokenIds[_treasure.claimedToken - 1]` in case of ERC721) in variable and pass it to event instead of _treasure.tokenId.
But you may also consider to remove `tokenId` field from the `Treasure` at all, as soon as you can still use `tokenIds` array to keep few id's or just one and it in the `TreasureTransferred` after.

Gas optimizations

Current average claim costs, tested as an average of 10 tx's for one user, numbers in GWEI:

First claim tx	Second claim tx	10 claims in total
195337	238939	2345788

1. It looks like you don't plan to allocate additional claim spots during the week, according to the logic provided. There is a `WinnerUpdationPeriod` and `ClaimPeriod` that follows after it. In this case, it's possible to make people claim all tokens at once in one contract transaction, which will reduce the costs for calling the contract each time and also make it possible to reuse some variables.

2. Claiming all tokens at once will allow us to get rid of some on-chain write operations, such as updating `week.winners[msg.sender].treasureTypeClaimed`. This will not be necessary since you won't need to keep this information between the calls; only one transaction is needed. You can keep this information as a memory array and loop through it every time you need to find if a specific type of treasure was claimed by a user or not. It will still be cheaper than using `week.winners[msg.sender].treasureTypeClaimed`, and the cost of it will increase with more treasure types to be claimed by a single user (which is not such a frequent case).

3. Executing all claims in one transaction will also require modifying the randomization logic a bit, by adding a unique variable for every claim cycle (otherwise, it will always be the same since block variables and `msg.sender` are the same). You can use the cycle index for these purposes in the `claim` function and `weekNumber` in the case of `updateWeeklyWinners`.

Claim Costs Table with Changes Above Implemented:

1 claim in total	2 claims in total	10 claims in total
198978	295266	1030181

This will allow reducing the gas costs by ~66% for a 10-token claim and ~32% when people claim two tokens at once. However, it will slightly increase the price of a 1-token claim transaction.

4. The `claimTreasure` function also calls the `PxChainlinkManager.isSignerVerifiedFromSignature`, which not only sounds like a poor logical division but also adds additional costs for calling the external contract. Moving this function to `PxUtils` will follow the good logical division practice and also decrease the one-time call cost to an average of **168,422** GWEI for a 1-token claim (~14% less than the initial one).

Final claim costs table with all changes implemented:

1 claim in total	2 claims in total	10 claims in total
168422	284654	1015233

That's all changes that can be performed to significantly decrease the gas costs without reworking the contract's structure, note that the real gas savings may vary from case to case depending on the amount of `Treasures` allocated for the week, average tokens to be claimed per user, etc.

Everything above was tested with 10 types of `Treasures` with 1:1 proportion of ERC1155 and ERC721.

You may also want to revise the data you store onchain and if you really need it.

Currently, the contract uses Chainlink randomization to select winners only, but not for determining the prizes. Instead, prize randomization is done fully on-chain. This approach not only lacks true randomness and has the potential to be predicted, but it also increases the gas costs.

To reduce the gas costs even further, you could consider moving the prize randomization process to your backend.

You may find a code sample in attachment, **please note that's just a reference for the changes described above and should not be used in production without proper refining due it's lack of checks.**

Please also note that SponsoredTrip claim was ommitted in provided examples, do not forget to add it in case if you will use it as reference for the adjustments.