PixelmonEvolution 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:

- **1 contract** was provided in total
    - PixelmonEvolution
- **Simple contracts logic description**
- **Smart contract list that interacts with the evolution smart contract**

The contracts code was provided on 12th April 2023. Audit report was delivered on 17th April 2023. It was given in the following formats:

- **PDF document**

Disclaimer

Summary

The contract is well-structured and written. The use of EIP712 is particularly noteworthy, as it demonstrates adherence to best practices in the field. No logical flaws or vulnerabilities were found throughout the audit process, and as such, the corresponding sections of the report have been omitted.

Please note that contracts Pixelmon and Serum contracts are out of scope of this audit.

In the following section, recommendations are provided for significantly reducing the gas cost of the primary functions.

Findings summary

	Critical	Medium	Low
PixelmonEvolution	0	0	0
Total	0	0	0

Gas optimizations

Main gas consumers:

- **claimPixelmonToken**
    - evolved/unevolved pixelmon token transfer from contract
- **evolvePixelmon**
    - transferring serum tokens to 0xdEaD
    - evolved pixelmon mint
    - transferring unevolved pixelmon token to contract
    - storing `StakedTokenInformation`

claimPixelmon

Evolved/unevolved pixelmon token transfer from contract

`PIXELMON_CONTRACT.safeTransferFrom(address(this), msg.sender, tokenId, "");`

`safeTransferFrom` is used to avoid situations when a token will be accidentally transferred to a contract that cannot handle this type of tokens.

If you don't plan for this function to be called from contracts, you can add the `noContracts` modifier and replace `safeTransferFrom` with `transferFrom`.

evolvePixelmon

There is not much we can do with the first two consumers.

Transferring unevolved pixelmon token to contract

`PIXELMON_CONTRACT.safeTransferFrom(msg.sender, address(this), tokenId);`

`safeTransferFrom` is used to avoid situations when a token will be accidentally transferred to a contract that cannot handle this type of tokens. In our case, we transfer the token from the user to our own contract, which we know can handle it.

Simply replacing it with `transferFrom` will save you **~4000 GWEI** for every token.

Storing `StakedTokenInformation`

```
struct StakedTokenInformation {
        uint256 tokenId;
        address owner;
        uint256 stakedAt;
        uint256 stakedFor;
}
```

Writing data to the contract is one of the most expensive actions. This struct will be stored twice per `evolvePixelmon` transaction. In Solidity, data is stored using 32 bytes per slot. If a slot is not fully occupied (less than 32 bytes variable stored), we can use the space left in the slot to store one more variable.

In the case of the structure above, it will occupy 4 slots:

- uint256 tokenId; - 32 bytes
- address owner; - 20 bytes
- uint256 stakedAt; - 32 bytes
- uint256 stakedFor; - 32 bytes

We can reduce the space occupied and fit this structure into 1 slot only:

```
struct StakedTokenInformation {
        address owner;
        uint32 tokenId;
        uint32 stakedAt;
        uint32 stakedFor;
}
```

Since the max value of uint32 is 4294967295, we can safely use it to store `tokenId`. For the timestamps - it will work as long as the `stakedAt` timestamp is less than the max value (token staked prior to **February 7, 2106**) and it's staked for less than **~136 years**.

Optimizing the structure in this way will save **~95000 GWEI** per token evolved.

Please note that this optimization will require you to add some casts to your code from **uint256** to **uint32** and also change the methods' arguments. In some cases, it will slightly increase the gas cost for casting uint256 to uint32, but it's entirely negligible compared to the savings mentioned above.

Here is a small comparison of the gas cost for `evolvePixelmon` after the changes above.

	Original (GWEI)	Optimized (GWEI)	Savings (%)
1 token evolution	355 897	260 566	26.68
5 tokens evolution	1 212 198	742 057	38.79

Minor gas optimization tips

- use `index = _uncheckedInc(index)` instead if `index++` in loops where it's possible

```
// evolvePixelmon
for (uint256 index = 0; index < serumAmounts.length; index++) {
        totalEvolveAmount += serumAmounts[index];
}
```

- `++variable` consumes a bit less gas than `variable++`

```
// evolvePixelmon
unchecked {
        nonces[msg.sender]++;
}
...
unchecked {
        nextEvolvePixelmonId++;
}
```