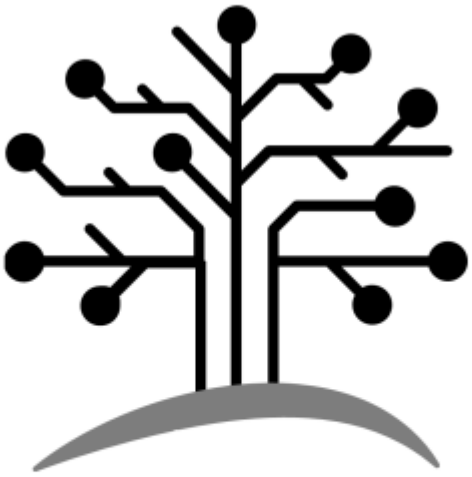

DeFiesta

Blue Trading

SECURITY REVIEW

Date: 22 August 2023

CONTENTS

1. About Defiesta

We are Defiesta — a company on a mission to make web3 protocols more secure, cost-efficient and user-friendly. Our team boasts extensive experience in the web3 space as both smart contract auditors and developers that have worked on top 100 blockchain projects with multi-million dollars in market capitalization.

Book an audit and learn more about us at defiesta.net or @DefiestaAudits

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Defiesta against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Blue Trading

Blue Trading is a trading card game incorporating NFT cards accompanied by original stories. NFT holders can access exclusive tales penned by a fantasy author, delving into the lore of Blue Trading and offering insights into characters and their worlds. The game combines rare card collecting with professionally written narratives, creating an immersive experience for fans of fantasy literature and trading card games. The stories are carefully crafted by an author for authenticity and quality, making each one a unique piece of art.

4. Risk Classification

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

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired.

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors

- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Audit Summary

This is the first audit for the protocol's smart contract component, which represents a single NFT contract of the ERC-721 standard. Considering the small size codebase, the security review managed to identify issues, tackling the random number generation process and reentrancy, among other informational and gas optimization findings.

The Nat Spec is comprehensive. The code's readability could be further improved via the implementation of the Informational findings (not included in the report), which also outline some foundational best practices.

We extend our gratitude to the Blue Trading's blockchain team for their exemplary responsiveness, offering comprehensive clarifications and detailed responses to our inquiries.

We would also like to point out that the project's network of choice - IOTA's Shimmer, is still a relatively unexplored territory in terms of performance and network-level bugs and issues that might create additional attack surfaces.

5.1 Protocol Summary

Project Name	**Blue Trading**
Repository	BlueTrading
Type of Project	ERC-721 collection
Audit Timeline	3 days
Review Commit Hash	45d9a7fbccb7a647f649a3a14f9d3f2bfa1c5f73
Fixes Review Commit Hash	N/A

5.2 Scope

The following smart contracts were in the scope of audit:

File	**nSLOC**
contracts/BlueTrading.sol	143
Total	**143**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **0**
- **Medium** issues: **3**
- **Low** issues: **3**

ID	Title	Severity
[M-01]	Insecure Generation of Randomness Used for Token Determination Logic	Medium
[M-02]	Using the **transfer** Function of **address payable** is Discouraged	Medium
[M-03]	Centralization Risk Due to Trusted Owner	Medium
[L-01]	Missing Reentrancy Protection For **BlueTrading._mint()** Function	Low
[L-02]	Missing Zero Value Check for **_mintingCost** Might Lead to Loss of Funds	Low
[L-03]	Ownership Role Transfer Function Implement Single-Step Role Transfer	Low

7. Findings

[M-01] Insecure Generation of Randomness Used for Token Determination Logic

Severity

Medium Risk

Description

Generating a **uint256** random value that relies on variables like **block.timestamp**, **randomizationNonce** and **msg.sender** as a source of randomness is a common vulnerability, as the outcome can be influenced/predicted by miners/validators, but even normal users can easily replicate these three sources of entropy:

1. **block.timestamp** can be replicated inside an attacker contract if the attack transaction is included in the same block as the PRNG.
2. **randomizationNonce** can be predicted since the value is deterministically incremented by 1, and the previous values are publicly accessible (like any state variable of a smart contract on a public blockchain).
3. **msg.sender** can be replicated since it's simply the attacker contract's address.

Location of Affected Code

File: contracts/BlueTrading.sol#L264-L265

```
uint256 randomNumber = uint256(keccak256(abi.encodePacked(block.timestamp, randomizationNonce,
msg.sender)));
uint256 randomIndex =randomNumber % tokenIdsToStartMinting.length;
```

Recommendations

Consider using a decentralized oracle for the generation of random numbers, such as Chainlink VRF. It is important to take into account the **requestConfirmations** variable that will be used in the VRFv2Consumer contract when implementing VRF. The purpose of this value is to specify the minimum number of blocks you wish to wait before receiving randomness from the Chainlink VRF service.

Team Response

Acknowleged.

[M-02] Using the transfer function of address payable is discouraged

Severity

Medium Risk

Description

The **transfer()** function only allows the recipient to use **2300** gas. If the recipient uses more than that, transfers will fail. This could, for example, be the case if **vendor** is the address of a multisig or payment splitter that is supposed to execute additional logic after the withdrawal. Furthermore, gas costs might change in the future, increasing the likelihood of that happening. Also, notice that **vendor** is immutable after deployment.

Consider the following scenario:

1. During deployment, Blue Trading is not aware of this "**transfer()** issue" and sets **vendor** to a contract address (e.g., a payment splitter) that consumes more than **2300** gas.
2. Blue Trading launches the collection.
3. Users mint the entire collection.
4. Blue Trading tries to withdraw the revenue from the contract. However, all of their attempts revert because the recipient consumes more than **2300** gas when receiving the funds.

Notice that Blue Trading cannot reset **vendor** to another address (e.g., an externally owned account) because the contract does not provide such functionality. Therefore, all the revenue that Blue Trading earned during the mint of the collection is stuck in the contract forever.

Description

File: contracts/BlueTrading.sol/L230

```
payable(vendor).transfer(address(this).balance);
```

Recommendations

While this issue will never occur as long as vendor represents an externally owned account, Blue Trading might want to set vendor to a contract address during deployment. Therefore, we recommend using call() instead of transfer() to withdraw the contract's SMR balance because call() will forward all available gas instead of only 2300 gas.

```
(bool success, ) = msg.sender.call{value: address(this).balance}("");
require(success, "withdraw failed.");
```

Team Response

Acknowleged and fixed.

[M-03] Centralization Risk Due to Trusted Owner

Severity

Medium Risk

Description

The contract has an owner with the privileged right to pause and unpause most of the contract's functionality and therefore it needs to be trusted. Currently, the contract owner is not prevented from renouncing the ownership while the contract is paused, which could cause any user assets stored in the protocol, to be locked indefinitely.

Both **Blue Trading** and **Defiesta** have been clear from the get-go that this functionality was only implemented because **EU** law (Article 30 of REPORT on the proposal for a regulation) currently requires **that a mechanism exists to terminate the continued execution of transactions.** Nonetheless, Blue Trading kindly asked Defiesta to incorporate this (hopefully temporary) issue here in the report to ensure Blue Trading's users enjoy full transparency. **Blue Trading intends to keep this privileged role only as long as the legal situation is not fully clarified. Furthermore, Blue Tra ding has implemented a dedicated function to revoke their privileged role in the contract as soon as they are certain that they don't violate EU law by doing so.** Blue Trading will make an effort to investigate this issue further and stay up to date with the latest legal developments.

Location of Affected Code

File: contracts/BlueTrading.sol#L163

File: contracts/BlueTrading.sol#L187

```
function mintBluk(uint256 _bulkAmount) external payable whenNotPaused {
```

Recommendation

It is recommended that the client carefully manages the private key of the controller account to avoid any potential hacking risk. Measures that can be taken are to enhance centralized privileges and roles in the protocol through a decentralized mechanism or module-based accounts with enhanced security practices. We propose to make the owner of **Blue Trading** a multi-sig wallet behind a **Timelock** contract so that users can monitor what transactions are about to be executed by this account and take action if necessary.

Team Response

Acknowleged.

[L-01] Missing Reentrancy Protection For BlueTrading._mint() Function

Severity

Low Risk

Description

A potential threat emerges from the **mint()** function, as it internally calls **safeMint()**, which triggers the **onERC721Received** callback. This could potentially execute malicious code, allowing an attacker to claim all tokens. It's important to be aware that the **mintingCost** will be paid for each iteration.

Location of Affected Code

File: contracts/BlueTrading.sol#L146-149

```
for (uint256 i = tokenIdToStartMinting; i < tokenIdToStartMinting + numberOfTokensPerMint; i++) {
        _safeMint(msg.sender, i);
        mintedTokenIds[i - tokenIdToStartMinting] = i;
    }
```

Recommendation

To protect against cross-function reentrancy attacks, OpenZeppelin's **nonReentrant** modifier that guards the decorated function with a mutex against reentrancy attacks should be applied.

```
import {ReentrancyGuard} from "@openzeppelin/contract/security/ReentrancyGuard.sol";

contract BlueTrading is ERC721, ERC721Enumrable, Pausable, Ownable, ReentrancyGuard {
```

```
function mint() external payable nonReentrant whenNotPaused {
```

It's also a best practice to follow the CEI (Checks-Effects-Interactions) pattern.

File: contracts/BlueTrading.sol#L163

```
for (uint256 i = tokenIdToStartMinting; i < tokenIdToStartMinting + numberOfTokensPerMint; i++) {
        mintedTokenIds[i - tokenIdToStartMinting] = i;
        _safeMint(msg.sender, i);
    }
```

Team Response

Acknowleged and fixed.

[L-02] Missing Zero Value Check for _mintingCost Might Lead to Loss of Funds

Severity

Low Risk

Description

The **_mintingCost** variable in the constructor is missing a zero-value check. This variable serves as a fundamental parameter in the protocol's operation, dictating the cost associated with minting tokens. If it is set to 0 by mistake, the protocol's business logic could be severely impacted, as there will be no minting tax fees.

Location of Affected Code

File: contracts/BlueTrading.sol#L77

```
constructor(
    string memory _name,
    string memory _symbol,
    string memory _baseURI_,
    uint256 _mintingCost,
    uint256 _numberOfTokensPerMint,
    uint256 _maxBulkBuy,
    uint256 _maxMints,
    uint256 _allowMintingAfter,
    address _vendor
) ERC721(_name, _symbol) {
    mintingCost = _mintingCost;
}
```

Recommendation

To address this vulnerability, consider adding a check so that it is not possible for **_mintingCost** to be **0.**

```
constructor(
    string memory _name,
    string memory _symbol,
    string memory _baseURI_,
    uint256 _mintingCost,
    uint256 _numberOfTokensPerMint,
    uint256 _maxBulkBuy,
    uint256 _maxMints,
    uint256 _allowMintingAfter,
    address _vendor
) ERC721(_name, _symbol) {
    require(_mintingCost != 0, "must not equal 0");
    mintingCost = _mintingCost;
}
```

Team Response

Acknowleged and fixed.

[L-03] Ownership Role Transfer Function Implement Single-Step Role Transfer

Severity

Low Risk

Description

The current ownership transfer process for all the contracts inheriting from **Ownable** involves the current owner calling the **transferOwnership()** function. If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing access to all functions with the **onlyOwner** modifier.

Location of Affected Code

File: contracts/BlueTrading.sol#L24

Recommendation

It is recommended to implement a two-step process where the owner nominates an account and the nominated account needs to call an **acceptOwnership()** function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. This can be easily achieved by using OpenZeppelin's **Ownable2Step** contract instead of **Ownable.**

Team Response

Acknowledged and fixed.