SMART CONTRACT AUDIT REPORT

for

Boson Protocol

Prepared By: Xiaomi Huang

PeckShield

September 9, 2022

Document Properties

Client	Boson Protocol
Title	Smart Contract Audit Report
Target	Boson Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 9, 2022	Luck Hu	Final Release
1.0-rc	September 5, 2022	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Boson protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Boson Protocol

The Boson protocol is a decentralized optimistic fair exchange protocol, which enables the trust-minimized, automated exchange of off-chain assets, whilst tokenizing commitments to trade as redeemable NFTs. The protocol enables the creation of a single digital market for physical assets, built on decentralized infrastructure and without the need for centralized intermediaries to enable fair exchange. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Boson Protocol

Item	Description
Name	Boson Protocol
Website	https://www.bosonprotocol.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 9, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/bosonprotocol/boson-protocol-contracts.git (25ea648)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/bosonprotocol/boson-protocol-contracts.git (0be076d)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Boson` protocol smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	3	■■■
Informational	0	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Boson Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Bypassed authorizeCommit() in onVoucherTransferred()	Business Logic	Confirmed
PVE-002	Low	Revisited supplyAvailable Checks in transferTwins()	Coding Practices	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Medium	Proper Return of Twins For Refused State	Business Logic	Confirmed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Revisited Twins Range Validation in createTwinInternal()	Coding Practices	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Bypassed authorizeCommit() in onVoucherTransferred()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ExchangeHandlerFacet`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `Boson` protocol, when a buyer agrees to the terms of the offer, he/she may proceed to commit the offer and receive a voucher which is an `RedeemableNFT` (`rNFT`) that is redeemable for the off-chain asset. The buyer may then choose either to transfer or trade the voucher, before ultimately moving to the `Redemption` phase. While examining the voucher transfer in the `onVoucherTransferred()` routine, we notice the new buyer may not be properly authorized to commit the offer.

To elaborate, we show below the code snippets of the `commitToOffer()/onVoucherTransferred()` routines. As the name indicates, the `commitToOffer()` routine is used for the buyer to commit an offer. Specially, if the offer is in a conditional group, it calls the `authorizeCommit()` routine to check whether the buyer is authorized to commit the offer (line 92). If the buyer is not authorized, he/she can not commit the input offer. Otherwise, the buyer can commit the offer and receive a voucher. Similarly, if the buyer wants to transfer the voucher to a new buyer, the new buyer shall also be authorized by the group condition. However, in the `onVoucherTransferred()` routine, it does not invoke the `authorizeCommit()` to check whether the new buyer is authorized or not. As a result, an authorized buyer could commit the offer for any un-authorized buyer.

```
64     function commitToOffer(address payable _buyer, uint256 _offerId)
65         external
66         payable
67         override
68         exchangesNotPaused
69         buyersNotPaused
```

```
        nonReentrant
    {
        // Make sure buyer address is not zero address
        require(_buyer != address(0), INVALID_ADDRESS);

        // Get the offer
        bool exists;
        Offer storage offer;
        (exists, offer) = fetchOffer(_offerId);

        // Make sure offer exists, is available, and isn't void, expired, or sold out
        require(exists, NO_SUCH_OFFER);

        OfferDates storage offerDates = fetchOfferDates(_offerId);
        require(block.timestamp >= offerDates.validFrom, OFFER_NOT_AVAILABLE);
        require(!offer.voided, OFFER_HAS_BEEN_VOIDED);
        require(block.timestamp < offerDates.validUntil, OFFER_HAS_EXPIRED);
        require(offer.quantityAvailable > 0, OFFER_SOLD_OUT);

        uint256 exchangeId = protocolCounters().nextExchangeId++;

        // Authorize the buyer to commit if offer is in a conditional group
        require(authorizeCommit(_buyer, offer, exchangeId), CANNOT_COMMIT);
        ...
    }
```

Listing 3.1: ExchangeHandlerFacet::commitToOffer()

```
    function onVoucherTransferred(uint256 _exchangeId, address payable _newBuyer)
        external
        override
        buyersNotPaused
        nonReentrant
    {
        // Get the exchange, should be in committed state
        (Exchange storage exchange, Voucher storage voucher) = getValidExchange(
            _exchangeId, ExchangeState.Committed);

        // Make sure that the voucher is still valid
        require(block.timestamp <= voucher.validUntilDate, VOUCHER_HAS_EXPIRED);

        (, Offer storage offer) = fetchOffer(exchange.offerId);

        // Make sure that the voucher was issued on the clone that is making a call
        require(msg.sender == protocolLookups().cloneAddress[offer.sellerId],
            ACCESS_DENIED);

        // Decrease voucher counter for old buyer
        protocolLookups().voucherCount[exchange.buyerId]--;
        // Fetch or create buyer
        uint256 buyerId = getValidBuyer(_newBuyer);

        // Update buyer id for the exchange
```

```
        exchange.buyerId = buyerId;

        // Increase voucher counter for new buyer
        protocolLookups().voucherCount[buyerId]++;

        // Notify watchers of state change
        emit VoucherTransferred(exchange.offerId, _exchangeId, buyerId, msgSender());
    }
```

Listing 3.2: ExchangeHandlerFacet::onVoucherTransferred()

Recommendation Properly invoke the `authorizeCommit()` in the `onVoucherTransferred()` routine to ensure the new buyer is authorized to take the voucher.

Status The issue has been confirmed by the team to be as designed. The team clarified that: at this point, the offer has already been committed to by the initial buyer who meets the condition. A secondary market buyer who does not meet the condition would have no way to know they were buying a voucher they could not use. In that situation, they would have no recourse to get back their money from the initial buyer. So, conditional commit only applies to the initial buyer.

3.2 Revisited supplyAvailable Checks in transferTwins()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ExchangeHandlerFacet`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

The `Boson` protocol supports a `phygital` module which makes it possible to link a physical thing to its digital counterpart (hence making a `phygital` twin) and furthermore to bundle multiple items or twins within a single offer. The `phygital` module allows the creation of an offer for an off-chain asset along with a number of `ERC-721`, `ERC-1155` and `ERC-20` tokens. When the voucher of a committed offer is redeemed, the linked twins are transferred from the seller to the buyer.

To elaborate, we show below the code snippet of the `transferTwins()` routine which is used to transfer the twins to the buyer. For each twin, before transferring the token(s), it decrements the transferred amount from the twin supply if the supply is limited (lines $639-644$). However, it comes to our attention that there are redundant validations for the supply and the transferred amount (lines $646, 656, 674$) which may fail and revert the transfer (because the transferred amount has been decremented from the supply).

Based on this, we suggest to remove these redundant validations in `transferTwins()`.

```
function transferTwins(Exchange storage _exchange, Voucher storage _voucher)
    internal
    returns (bool shouldBurnVoucher)
{
    // See if there is an associated bundle
    (bool exists, uint256 bundleId) = fetchBundleIdByOffer(_exchange.offerId);

    // Voucher should be burned in the happy path
    shouldBurnVoucher = true;

    // Transfer the twins
    if (exists) {
        // Get storage location for bundle
        (, Bundle storage bundle) = fetchBundle(bundleId);

        // Get the twin Ids in the bundle
        uint256[] storage twinIds = bundle.twinIds;

        // Get seller account
        (, Seller storage seller, ) = fetchSeller(bundle.sellerId);

        address sender = msgSender();
        // Variable to track whether some twin transfer failed
        bool transferFailed;

        uint256 exchangeId = _exchange.id;

        // Visit the twins
        for (uint256 i = 0; i < twinIds.length; i++) {
            // Get the twin
            (, Twin storage twin) = fetchTwin(twinIds[i]);

            // Transfer the token from the seller's operator to the buyer
            // N.B. Using call here so as to normalize the revert reason
            bytes memory result;
            bool success;
            uint256 tokenId = twin.tokenId;
            TokenType tokenType = twin.tokenType;

            // Shouldn't decrement supply if twin supply is unlimited
            if (twin.supplyAvailable != type(uint256).max) {
                // Decrement by 1 if token type is NonFungible otherwise decrement
                    amount (i.e, tokenType is MultiToken or FungibleToken)
                twin.supplyAvailable = twin.tokenType == TokenType.NonFungibleToken
                    ? twin.supplyAvailable - 1
                    : twin.supplyAvailable - twin.amount;
            }

            if (tokenType == TokenType.FungibleToken && twin.supplyAvailable >= twin
                .amount) {
                // ERC-20 style transfer
```

```
                    (success, result) = twin.tokenAddress.call(
                        abi.encodeWithSignature(
                            "transferFrom(address,address,uint256)",
                            seller.operator,
                            sender,
                            twin.amount
                        )
                    );
                } else if (tokenType == TokenType.NonFungibleToken && twin.supplyAvailable > 0) {
                    // Token transfer order is ascending to avoid overflow when twin supply is unlimited
                    if (twin.supplyAvailable == type(uint256).max) {
                        twin.tokenId++;
                    } else {
                        // Token transfer order is descending
                        tokenId = twin.tokenId + twin.supplyAvailable;
                    }
                    // ERC-721 style transfer
                    (success, result) = twin.tokenAddress.call(
                        abi.encodeWithSignature(
                            "safeTransferFrom(address,address,uint256,bytes)",
                            seller.operator,
                            sender,
                            tokenId,
                            ""
                        )
                    );
                } else if (twin.tokenType == TokenType.MultiToken && twin.supplyAvailable >= twin.amount) {
                    // ERC-1155 style transfer
                    (success, result) = twin.tokenAddress.call(
                        abi.encodeWithSignature(
                            "safeTransferFrom(address,address,uint256,uint256,bytes)",
                            seller.operator,
                            sender,
                            tokenId,
                            twin.amount,
                            ""
                        )
                    );
                }
                ...
        }
    }
```

Listing 3.3: ExchangeHandlerFacet::transferTwins()

Recommendation Remove the redundant validations for the transferred amount and the supply in `transferTwins()`.

Status The issue has been fixed by this commit: `98fb82a`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ExchangeHandlerFacet`, `FundsLib`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transferFrom()`, there is a check, i.e., `if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }
    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: `ZRX.sol`

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

In the following, we show the `ExchangeHandlerFacet::transferTwins()` routine. If the `ZRX` token is supported as `twin.tokenAddress`, the call to the unsafe version of `transferFrom(address,address,uint256)` (line 650) may return `false` while not revert for token transfer failure. In this case, there is a need to check whether the `result` is `true` or `false`. Note the transfer is successful only when the `abi.decode(result, (bool))== true` or there is no return data (`result.length == 0`).

```
599     function transferTwins(Exchange storage _exchange, Voucher storage _voucher)
600         internal
601         returns (bool shouldBurnVoucher)
602     {
603         // See if there is an associated bundle
604         (bool exists, uint256 bundleId) = fetchBundleIdByOffer(_exchange.offerId);
605 
606         // Voucher should be burned in the happy path
607         shouldBurnVoucher = true;
608 
609         // Transfer the twins
610         if (exists) {
611             // Get storage location for bundle
612             (, Bundle storage bundle) = fetchBundle(bundleId);
613 
614             // Get the twin Ids in the bundle
615             uint256[] storage twinIds = bundle.twinIds;
616 
617             // Get seller account
618             (, Seller storage seller, ) = fetchSeller(bundle.sellerId);
619 
620             address sender = msgSender();
621             // Variable to track whether some twin transfer failed
622             bool transferFailed;
623 
624             uint256 exchangeId = _exchange.id;
625 
626             // Visit the twins
627             for (uint256 i = 0; i < twinIds.length; i++) {
628                 // Get the twin
629                 (, Twin storage twin) = fetchTwin(twinIds[i]);
630 
631                 // Transfer the token from the seller's operator to the buyer
632                 // N.B. Using call here so as to normalize the revert reason
633                 bytes memory result;
634                 bool success;
635                 uint256 tokenId = twin.tokenId;
636                 TokenType tokenType = twin.tokenType;
637 
638                 // Shouldn't decrement supply if twin supply is unlimited
639                 if (twin.supplyAvailable != type(uint256).max) {
```

```
            // Decrement by 1 if token type is NonFungible otherwise decrement amount (i.e, tokenType is MultiToken or FungibleToken)
            twin.supplyAvailable = twin.tokenType == TokenType.NonFungibleToken
                ? twin.supplyAvailable - 1
                : twin.supplyAvailable - twin.amount;
        }

        if (tokenType == TokenType.FungibleToken && twin.supplyAvailable >= twin.amount) {
            // ERC-20 style transfer
            (success, result) = twin.tokenAddress.call(
                abi.encodeWithSignature(
                    "transferFrom(address,address,uint256)",
                    seller.operator,
                    sender,
                    twin.amount
                )
            );
        } else if (tokenType == TokenType.NonFungibleToken && twin.supplyAvailable > 0) {
            // Token transfer order is ascending to avoid overflow when twin supply is unlimited
            if (twin.supplyAvailable == type(uint256).max) {
                twin.tokenId++;
            } else {
                // Token transfer order is descending
                tokenId = twin.tokenId + twin.supplyAvailable;
            }
            // ERC-721 style transfer
            (success, result) = twin.tokenAddress.call(
                abi.encodeWithSignature(
                    "safeTransferFrom(address,address,uint256,bytes)",
                    seller.operator,
                    sender,
                    tokenId,
                    ""
                )
            );
        } else if (twin.tokenType == TokenType.MultiToken && twin.supplyAvailable >= twin.amount) {
            // ERC-1155 style transfer
            (success, result) = twin.tokenAddress.call(
                abi.encodeWithSignature(
                    "safeTransferFrom(address,address,uint256,uint256,bytes)",
                    seller.operator,
                    sender,
                    tokenId,
                    twin.amount,
                    ""
                )
            );
        }
```

```

                // If token transfer failed
                if (!success) {
                    transferFailed = true;

                    emit TwinTransferFailed(twin.id, twin.tokenAddress, exchangeId, tokenId, twin.amount, sender);
                } else {
                    // Store twin receipt on twinReceiptsByExchange
                    protocolLookups().twinReceiptsByExchange[exchangeId].push(
                        TwinReceipt(twin.id, tokenId, twin.amount, twin.tokenAddress, twin.tokenType)
                    );

                    emit TwinTransferred(twin.id, twin.tokenAddress, exchangeId, tokenId, twin.amount, sender);
                }
            }

            if (transferFailed) {
                // Raise a dispute if caller is a contract
                if (isContract(sender)) {
                    raiseDisputeInternal(_exchange, _voucher, seller.id);
                } else {
                    // Revoke voucher if caller is an EOA
                    revokeVoucherInternal(_exchange);
                    // N.B.: If voucher was revoked because transfer twin failed, then voucher was already burned
                    shouldBurnVoucher = false;
                }
            }
        }
    }
```

Listing 3.5: `ExchangeHandlerFacet::transferTwins()`

Note another routine, i.e., `FundsLib::transferFundsFromProtocol()`, can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()` and `transferFrom()`.

Status The issue has been fixed by this commit: `61715ce`.

3.4 Proper Return of Twins For Refused State

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: FundsLib
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.2, the Boson protocol supports a phygital module which allows the creation of an offer for an off-chain asset along with a number of ERC-721, ERC-1155 and ERC-20 tokens. When the voucher of the committed offer is redeemed, the linked twins are transferred to the buyer from the seller.

After redeeming the voucher, if the buyer asserts that the seller has failed to meet their obligations within the agreement, the buyer can raise dispute to push the protocol into the Disputed state. Furthermore, the buyer can seek escalated dispute resolution from the dispute resolver (DR). The DR will analyze the case offline, and provide a decision for the split of escrowed funds that is to be refunded to each party. Specially, if the DR fails to provide the split for the dispute in the given time or explicitly refuses to decide on the dispute, the protocol ends up in the Refused state where the funds committed are reverted to the original parties as per the original offer.

To elaborate, we show below the code snippet of the releaseFunds() routine, which is used to refund the splitted escrowed funds to each party. When the protocol ends up in the Refused state (lines 174-177), the routine reverts the committed funds to the original parties as per the original offer. However, it does not return the linked twins back to the seller. The linked twins have been transferred to the buyer when the voucher is redeemed. When the committed funds are reverted in the Refused state, the twins shall also be returned to the seller.

```
116  function releaseFunds(uint256 _exchangeId) internal {
117    // Load protocol entities storage
118    ProtocolLib.ProtocolEntities storage pe = ProtocolLib.protocolEntities();
119
120    // Get the exchange and its state
121    // Since this should be called only from certain functions from exchangeHandler and
         disputeHandler
122    // exhange must exist and be in a completed state, so that's not checked explicitly
123    BosonTypes.Exchange storage exchange = pe.exchanges[_exchangeId];
124
125    // Get offer from storage to get the details about sellerDeposit, price, sellerId,
         exchangeToken and buyerCancelPenalty
126    BosonTypes.Offer storage offer = pe.offers[exchange.offerId];
127    // calculate the payoffs depending on state exchange is in
128    uint256 sellerPayoff;
```

```
    uint256 buyerPayoff;
    uint256 protocolFee;
    uint256 agentFee;

    BosonTypes.OfferFees storage offerFee = pe.offerFees[exchange.offerId];

    {
        // scope to avoid stack too deep errors
        BosonTypes.ExchangeState exchangeState = exchange.state;
        uint256 sellerDeposit = offer.sellerDeposit;
        uint256 price = offer.price;

        if (exchangeState == BosonTypes.ExchangeState.Completed) {
            // COMPLETED
            protocolFee = offerFee.protocolFee;
            // buyerPayoff is 0
            agentFee = offerFee.agentFee;
            sellerPayoff = price + sellerDeposit - protocolFee - agentFee;
        } else if (exchangeState == BosonTypes.ExchangeState.Revoked) {
            // REVOKED
            // sellerPayoff is 0
            buyerPayoff = price + sellerDeposit;
        } else if (exchangeState == BosonTypes.ExchangeState.Canceled) {
            // CANCELED
            uint256 buyerCancelPenalty = offer.buyerCancelPenalty;
            sellerPayoff = sellerDeposit + buyerCancelPenalty;
            buyerPayoff = price - buyerCancelPenalty;
        } else if (exchangeState == BosonTypes.ExchangeState.Disputed) {
            // DISPUTED
            // determine if buyerEscalationDeposit was encumbered or not
            // if dispute was escalated, disputeDates.escalated is populated
            uint256 buyerEscalationDeposit = pe.disputeDates[_exchangeId].escalated > 0
                ? pe.disputeResolutionTerms[exchange.offerId].buyerEscalationDeposit
                : 0;

            // get the information about the dispute, which must exist
            BosonTypes.Dispute storage dispute = pe.disputes[_exchangeId];
            BosonTypes.DisputeState disputeState = dispute.state;

            if (disputeState == BosonTypes.DisputeState.Retracted) {
                // RETRACTED - same as "COMPLETED"
                protocolFee = offerFee.protocolFee;
                agentFee = offerFee.agentFee;
                // buyerPayoff is 0
                sellerPayoff = price + sellerDeposit - protocolFee - agentFee +
                    buyerEscalationDeposit;
            } else if (disputeState == BosonTypes.DisputeState.Refused) {
                // REFUSED
                sellerPayoff = sellerDeposit;
                buyerPayoff = price + buyerEscalationDeposit;
            } else {
                // RESOLVED or DECIDED
```

```
                uint256 pot = price + sellerDeposit + buyerEscalationDeposit;
                buyerPayoff = (pot * dispute.buyerPercent) / 10000;
                sellerPayoff = pot - buyerPayoff;
            }
        }
    }

    // Store payoffs to availablefunds and notify the external observers
    address exchangeToken = offer.exchangeToken;
    uint256 sellerId = offer.sellerId;
    uint256 buyerId = exchange.buyerId;
    address sender = EIP712Lib.msgSender();
    if (sellerPayoff > 0) {
        increaseAvailableFunds(sellerId, exchangeToken, sellerPayoff);
        emit FundsReleased(_exchangeId, sellerId, exchangeToken, sellerPayoff, sender);
    }
    if (buyerPayoff > 0) {
        increaseAvailableFunds(buyerId, exchangeToken, buyerPayoff);
        emit FundsReleased(_exchangeId, buyerId, exchangeToken, buyerPayoff, sender);
    }
    if (protocolFee > 0) {
        increaseAvailableFunds(0, exchangeToken, protocolFee);
        emit ProtocolFeeCollected(_exchangeId, exchangeToken, protocolFee, sender);
    }
    if (agentFee > 0) {
        // Get the agent for offer
        uint256 agentId = ProtocolLib.protocolLookups().agentIdByOffer[exchange.offerId
            ];
        increaseAvailableFunds(agentId, exchangeToken, agentFee);
        emit FundsReleased(_exchangeId, agentId, exchangeToken, agentFee, sender);
    }
}
```

Listing 3.6: FundsLib::releaseFunds()

Recommendation Return the twins back to the seller when the protocol ends up in the `Refused` state.

Status The issue has been confirmed by the team to be as designed. The team clarified that: the protocol transfers the twin to the buyer in the `redeemVoucher()` and the protocol never has custody of the twin, nor approval to transfer it back from the buyer. In case of a dispute, regardless of the finalisation path, the parties must discuss and manage any possible return of twins outside the protocol.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In Boson protocol, there is a privileged ADMIN that plays a critical role in governing and regulating the protocol-wide operations.

To elaborate, we show below the sensitive operations that are related to ADMIN. Specifically, it has the authority to activate a new dispute resolver who is privileged to provide a decision for the split of escrowed funds for the escalated dispute; set the flat protocol fee for exchanges in BOSON; set the protocol fee percentage, etc.

```
    function activateDisputeResolver(uint256 _disputeResolverId)
        external
        disputeResolversNotPaused
        onlyRole(ADMIN)
        nonReentrant
    {
        bool exists;
        DisputeResolver storage disputeResolver;

        //Check Dispute Resolver and Dispute Resolver Fees from  disputeResolvers and
            disputeResolverFees mappings
        (exists, disputeResolver, ) = fetchDisputeResolver(_disputeResolverId);

        //Dispute Resolver  must already exist
        require(exists, NO_SUCH_DISPUTE_RESOLVER);

        disputeResolver.active = true;

        emit DisputeResolverActivated(_disputeResolverId, disputeResolver, msgSender());
    }
```

Listing 3.7: DisputeResolverHandlerFacet::activateDisputeResolver()

```
    function setProtocolFeeFlatBoson(uint256 _protocolFeeFlatBoson) public override
        onlyRole(ADMIN) nonReentrant {
        // Store fee percentage
        protocolFees().flatBoson = _protocolFeeFlatBoson;

        // Notify watchers of state change
        emit ProtocolFeeFlatBosonChanged(_protocolFeeFlatBoson, msgSender());
```

```
    }
```

Listing 3.8: ConfigHandlerFacet::setProtocolFeeFlatBoson()

It would be worrisome if the `ADMIN` account is a plain EOA account. A multi-sig account could greatly alleviates this concern, though it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

Recommendation Promptly transfer the `ADMIN` privilege to the intended governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated by the team as they will follow best practices and use a multi-sig wallet for the `ADMIN` role.

3.6 Revisited Twins Range Validation in createTwinInternal()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TwinBase`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

As mentioned in Section 3.2, the `Boson` protocol supports a `phygital` module which makes it possible to link a physical thing to its digital counterpart, thus making a `phygital` twin, and furthermore to bundle multiple items or twins within a single offer.

To elaborate, we show below the code snippet of the `createTwinInternal()` routine. As the name indicates, it is used to create a new twin. Specially, if the token type is `NonFungibleToken`, the twin has to provide a start `tokenId` and an available supply. If `supplyAvailable == type(uint256).max`, the supply is unlimited. In this case, it gets all the seller twins that belong to the same token address of the new twin to validate if any two of them have unlimited supply since ranges may overlap each other (lines 70-78). However, the condition `if (currentTwin.supplyAvailable == type(uint256).max || _twin.supplyAvailable == type(uint256).max)` (line 75) can be hit if any twin has unlimited supply, thus making it impossible to create a twin with unlimited supply. Our analysis shows that the condition can be changed to `if (currentTwin.supplyAvailable == type(uint256).max && _twin.supplyAvailable == type(uint256).max)`.

```
33      function createTwinInternal(Twin memory _twin) internal {
```

```
        // get message sender
        address sender = msgSender();

        // get seller id, make sure it exists and store it to incoming struct
        (bool exists, uint256 sellerId) = getSellerIdByOperator(sender);
        require(exists, NOT_OPERATOR);

        // Protocol must be approved to transfer sellers tokens
        require(isProtocolApproved(_twin.tokenAddress, sender, address(this)), NO_TRANSFER_APPROVED);

        // Twin supply must exist and can't be zero
        require(_twin.supplyAvailable > 0, INVALID_SUPPLY_AVAILABLE);

        if (_twin.tokenType == TokenType.NonFungibleToken) {
            // Check if the token supports IERC721 interface
            require(contractSupportsInterface(_twin.tokenAddress, 0x80ac58cd), INVALID_TOKEN_ADDRESS);

            // If token is NonFungible amount should be zero
            require(_twin.amount == 0, INVALID_TWIN_PROPERTY);

            // Calculate new twin range [tokenId...lastTokenId]
            uint256 lastTokenId;
            uint256 tokenId = _twin.tokenId;
            if (_twin.supplyAvailable == type(uint256).max) {
                require(tokenId <= (1 << 255), INVALID_TWIN_TOKEN_RANGE); // if supply is "unlimited", starting index can be at most 2*255
                lastTokenId = type(uint256).max;
            } else {
                require(type(uint256).max - _twin.supplyAvailable >= tokenId, INVALID_TWIN_TOKEN_RANGE);
                lastTokenId = tokenId + _twin.supplyAvailable - 1;
            }

            // Get all seller twin ids that belong to the same token address of the new twin to validate if they have not unlimited supply since ranges can overlaps each other
            uint256[] storage twinIds = protocolLookups().twinIdsByTokenAddressAndBySeller[sellerId][
                _twin.tokenAddress
            ];

            for (uint256 i = 0; i < twinIds.length; i++) {
                // Get storage location for looped twin
                (, Twin storage currentTwin) = fetchTwin(twinIds[i]);

                // The protocol cannot allow two twins with unlimited supply and with the same token address because range overlaps with each other
                if (currentTwin.supplyAvailable == type(uint256).max  _twin.supplyAvailable == type(uint256).max) {
                    require(currentTwin.tokenAddress != _twin.tokenAddress,
```

```
                        INVALID_TWIN_TOKEN_RANGE);
                }
            }

            // Get all ranges of twins that belong to the seller and to the same token
                address of the new twin to validate if range is available
            TokenRange[] storage twinRanges = protocolLookups().twinRangesBySeller[
                sellerId][_twin.tokenAddress];

            // Checks if token range isn't being used in any other twin of seller
            for (uint256 i = 0; i < twinRanges.length; i++) {
                // A valid range has:
                // - the first id of range greater than the last token id (tokenId +
                    initialSupply - 1) of the looped twin or
                // - the last id of range lower than the looped twin tokenId (beginning
                    of range)
                require(tokenId > twinRanges[i].end  lastTokenId < twinRanges[i].start,
                    INVALID_TWIN_TOKEN_RANGE);
            }

            // Add range to twinRangesBySeller mapping
            protocolLookups().twinRangesBySeller[sellerId][_twin.tokenAddress].push(
                TokenRange(tokenId, lastTokenId));
            // Add twin id to twinIdsByTokenAddressAndBySeller mapping
            protocolLookups().twinIdsByTokenAddressAndBySeller[sellerId][_twin.
                tokenAddress].push(_twin.id);
        }
        ...
    }
```

Listing 3.9: `TwinBase::createTwinInternal()`

Recommendation Revisit the twin range check in `createTwinInternal()` to allow at most one twin with the same token address to have unlimited supply.

Status The issue has been confirmed by the team to be as designed. The team clarified that: the seller can not have two twins with the same `tokenAddress` if one of them is unlimited.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Boson` protocol which is a decentralized optimistic fair exchange protocol that enables the trust-minimized, automated exchange of off-chain assets, whilst tokenizing commitments to trade as redeemable `NFTs`. The protocol enables the creation of a single digital market for physical assets, built on decentralized infrastructure and without the need for centralized intermediaries to enable fair exchange. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.