SMART CONTRACT AUDIT REPORT

for

BZEROX, LLC

Prepared By: Shuxiao Wang

Hangzhou, China
Sep. 1, 2020

Document Properties

Client	bZeroX, LLC
Title	Smart Contract Audit Report
Target	bZx v2.0
Version	1.0-rc1
Author	Chiachih Wu
Auditors	Xuxian Jiang, Chiachih Wu, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	Sep. 1, 2020	Chiachih Wu	Release Candidate #1

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the source code of **bZx v2.0** smart contract, we in the report outline 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About bZx v2.0

The bZx protocol is a set of smart contracts running on top of the Ethereum blockchain. The protocol focuses on lending and margin trading similar to the dYdX protocol. There are three main tokens in the bZx system, iTokens, pTokens, and BZRX tokens. The bZx system of lending and borrowing depends on iTokens and pTokens, and when users lend or borrow money on bZx, their crypto assets go into or come out of global liquidity pools, which are pools of funds shared between many different exchanges. When lenders supply funds into the global liquidity pools, they automatically receive iTokens; When users borrow money to open margin trading positions, they automatically receive pTokens. The system is also designed to use the BZRX tokens, which are only used to pay fees on the network currently.

The basic information of bZx v2.0 is as follows:

Table 1.1: Basic Information of bZx v2.0

Item	Description
Issuer	bZeroX, LLC
Website	https://bzx.network/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Sep. 1, 2020

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

- https://github.com/bZxNetwork/contractsV2 (e0c7ec0)

1.2 About PeckShield

PeckShield Inc. [15] 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 OWASP Risk Rating Methodology [10]:

- 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.

Table 1.3: The Full List of Check Items

Category	Check Item
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

To evaluate the risk, we go through a list of check 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 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) [9], 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.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 Logics	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 bZx v2.0 implementation. During the first phase of our audit, we studied the smart contract source code and ran 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	1	■
High	2	■ ■
Medium	2	■ ■
Low	4	■ ■ ■ ■
Informational	7	■ ■ ■ ■ ■ ■ ■
Total	16	

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 1 critical-severity vulnerability, 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 7 informational recommendations.

Table 2.1: Key bZx v2.0 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Business Logic Error in _burnToken()	Business Logics	Fixed
PVE-002	Low	Denial-of-Service Risk in borrow()	Business Logics	Fixed
PVE-003	High	Business Logic Error in marginTrade()	Business Logics	Fixed
PVE-004	Info.	Incompatible _dsrWithdraw() Return Value	Coding Practices	Fixed
PVE-005	Info.	Incessive _dsrDeposit() Call in _mintToken()	Coding Practices	Fixed
PVE-006	Info.	Zero Amount Flash Loan	Business Logics	Fixed
PVE-007	Critical	Confused Deputy in borrow()/marginTrade()	Business Logics	Fixed
PVE-008	Medium	Business Logic Error in getLoanParamsList()	Business Logics	Fixed
PVE-009	Medium	Inconsistent Fee Calculation in getBorrowAmount() and getRequiredCollateral()	Business Logics	Fixed
PVE-010	Info.	Reentrancy Risk in withdrawAccruedInterest()	Security Features	Fixed
PVE-011	Info.	Unused Variables in _initializeLoan()/_closeLoan()	Coding Practices	Confirmed
PVE-012	Low	Inconsistent Book-Keeping Records/Events Data in _payFeeReward()	Business Logics	Fixed
PVE-013	Low	Incompatibility With Deflationary Tokens in swapExternal()	Business Logics	Fixed
PVE-014	Info.	Improved Arithmetic Operations	Business Logics	Fixed
PVE-015	Info.	Business Error in _updateCheckpoints	Business Logics	Fixed
PVE-016	Low	Business Logic Error in queryReturn()	Business Logics	Fixed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Business Logic Error in _burnToken()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: LoanTokenLogicDai
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In bZx v2.0, loan token holders could burn tokens to get their underlying assets back. In particular, the LoanToken minted by depositing Dai or Chai tokens could be cashed out by invoking the external function burn() or burnToChai(). When reviewing the implementation of the LoanTokenLogicDai contract, we notice that the burnToChai() function has a business logic error which could lead to transferring the underlying Chai tokens to a wrong address.

As shown in the following code snippets, the external function burnToChai() allows the caller (msg.sender) to burn burnAmount of loan tokens and get the underlying Chai tokens to the receiver.

```
    function burnToChai(
        address receiver,
        uint256 burnAmount)
        external
        nonReentrant
        returns (uint256 chaiAmountPaid)
    {
        return _burnToken(
            burnAmount,
            receiver,
            true // toChai
        );
    }
```

Listing 3.1: LoanTokenLogicDai.sol

The internal function `_burnToken()` calculates the amount of Chai to be withdrawn. However, the Chai tokens are `move()`'ed to the `msg.sender` instead of the `receiver` (line 337). Compared to the case of withdrawing Dai tokens (`toChai = false`), the Dai tokens are withdrawn from DSR to `receiver`, which is inconsistent to the `toChai = true` case.

```
329         if (toChai) {
330             _dsrDeposit();

332             IChai _chai = _getChai();
333             uint256 chaiBalance = _chai.balanceOf(address(this));

335             success = _chai.move(
336                 address(this),
337                 msg.sender,
338                 amountPaid
339             );

341             // get Chai amount withdrawn
342             amountPaid = chaiBalance
343                 .sub(_chai.balanceOf(address(this)));
344         } else {
345             success = _dsrWithdraw(amountPaid).transfer(
346                 receiver,
347                 amountPaid
348             );

350             _dsrDeposit();
351         }
```

Listing 3.2: LoanTokenLogicDai.sol

Recommendation Fix the `toChai` case by `move()`'ing Chai from `address(this)` to `receiver`.

Status This issue has been addressed by fixing the Chai receiver in this commit: 24510aa.

3.2 Denial-of-Service Risk in borrow()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LoanTokenLogicStandard`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In bZx v2.0, the `borrowOrTradeFromPool()` function in the bZx contract is the core of opening a new loan. As shown in the following code snippets, the `msg.value` should be zero when the `loanDataBytes` is empty (line 62). However, we found a path from the loan token contract to the `borrowOrTradeFromPool()` with an empty `loanDataBytes` but a non-zero `msg.value`, which leads to a denial-of-service vulnerability.

```
40     function borrowOrTradeFromPool(
41         bytes32 loanParamsId,
42         bytes32 loanId, // if 0, start a new loan
43         bool isTorqueLoan,
44         uint256 initialMargin,
45         address[4] calldata sentAddresses,
46             // lender: must match loan if loanId provided
47             // borrower: must match loan if loanId provided
48             // receiver: receiver of funds (address(0) assumes borrower address)
49             // manager: delegated manager of loan unless address(0)
50         uint256[5] calldata sentValues,
51             // newRate: new loan interest rate
52             // newPrincipal: new loan size (borrowAmount + any borrowed interest)
53             // torqueInterest: new amount of interest to escrow for Torque loan (determines initial loan length)
54             // loanTokenReceived: total loanToken deposit (amount not sent to borrower in the case of Torque loans)
55             // collateralTokenReceived: total collateralToken deposit
56         bytes calldata loanDataBytes)
57         external
58         payable
59         nonReentrant
60         returns (uint256 newPrincipal, uint256 newCollateral)
61     {
62         require(msg.value == 0 || loanDataBytes.length != 0, "loanDataBytes required with ether");
```

Listing 3.3: LoanOpenings.sol::borrowOrTradeFromPool()

The path starts from the `borrow()` function in the loan token contract. In line 177-184, `_borrowOrTrade()` is invoked with an empty `loanDataBytes`.

```
177         return _borrowOrTrade(
178             loanId,
179             withdrawAmount,
180             2 * 10**18, // leverageAmount (translates to 150% margin for a Torque loan)
181             collateralTokenAddress,
182             sentAddresses,
183             sentAmounts,
184             "" // loanDataBytes
185         );
```

Listing 3.4: LoanTokenLogicStandard.sol::borrow()

Inside `_borrowOrTrade()`, `msgValue` is set as the ether balance of the loan token contract when `msg.value` is not zero.

```
        uint256 msgValue;
        if (msg.value != 0) {
```

```
            msgValue = address(this).balance;
            if (msgValue > msg.value) {
                msgValue = msg.value;
            }
        }
```

Listing 3.5: LoanTokenLogicStandard.sol::_borrowOrTrade()

Later on, the `msgValue` is passed into the bZx contract with the empty `loanDataBytes`.

```
        (sentAmounts[1], sentAmounts[4]) = ProtocolLike(bZxContract).
            borrowOrTradeFromPool.value(msgValue)( // newPrincipal, newCollateral
            loanParamsId,
            loanId,
            withdrawAmount != 0 ? // isTorqueLoan
                true :
                false,
            leverageAmount, // initialMargin
            sentAddresses,
            sentAmounts,
            loanDataBytes
        );
```

Listing 3.6: LoanTokenLogicStandard.sol::_borrowOrTrade()

This means the `borrow()` transaction would be always reverted if the loan token contract has some ether balance. Unfortunately, there's a public payable function, `flashBorrow()`, which allows an arbitrary user to intentionally leave some ether in the loan token contract. Those intentionally left ether would fail all the following `borrow()` calls.

Recommendation Fix the `msgValue` sent into the bZx contract.

Status This issue has been addressed by getting the `msgValue` from the `_verifyTransfers()` function which accurately compute the ether carried with the `borrow()` call in this commit: 24510aa.

3.3 Business Logic Error in marginTrade()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `LoanTokenLogicStandard`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

While tracing the code flow of `marginTrade()`, we notice that the implementation is incomplete when the execution reaches `_swapsCall()` with a non-empty `loanDataBytes`. As shown in the following code snippets, the `else` branch starting from line 133 leaves the system in an invalid state.

```
127         if (loanDataBytes.length == 0) {
128             (destTokenAmountReceived, sourceTokenAmountUsed) = _swapsCall_internal(
129                 addrs,
130                 vals
131             );
132         } else {
133             /*
134             //keccak256("Swaps_SwapsImplZeroX")
135             address swapsImplZeroX;
136             assembly {
137                 swapsImplZeroX := sload(0
                    x129a6cb350d136ca8d0881f83a9141afd5dc8b3c99057f06df01ab75943df952)
138             }
139             */
140             //revert(string(loanDataBytes));
141             /*
142             vaultWithdraw(
143                 addrs[0], // sourceToken
144                 address(zeroXConnector),
145                 sourceTokenAmount
146             );
147             (destTokenAmountReceived, sourceTokenAmountUsed) = zeroXConnector.swap.value
                (msg.value)(
148                 addrs[0], // sourceToken
149                 addrs[1], // destToken
150                 addrs[2], // receiver
151                 sourceTokenAmount,
152                 0,
153                 loanDataBytes
154             );
155             */
156         }
```

Listing 3.7: SwapsUser.sol :: _swapsCall()

Unfortunately, an user can set an arbitrary `loanDataBytes` in `marginTrade()` which leads the invalid state mentioned above.

```
        return _borrowOrTrade(
            loanId,
            0, // withdrawAmount
            leverageAmount,
            collateralTokenAddress,
            sentAddresses,
            sentAmounts,
            loanDataBytes
        );
```

Listing 3.8: LoanTokenLogicStandard.sol::marginTrade()

Recommendation Implement the `loanDataBytes.length != 0` case in `_swapsCall()`.

Status This issue has been addressed by reverting the `loanDataBytes.length != 0` case in `_swapsCall()` in this commit: 24510aa.

3.4 Incompatible _dsrWithdraw() Return Value

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LoanTokenLogicDai`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [3]

Description

The internal function, `_dsrWithdraw()`, in the `LoanTokenLogicDai` contract allows the caller to withdraw `_value` of Dai from the DSR for later usage. The `_dsrWithdraw()` function also returns the Dai address which seems a performance improvement as the Dai address is kept in the `_dai` local variable.

```
    function _dsrWithdraw(
        uint256 _value)
        internal
        returns (IERC20 _dai)
    {
        _dai = _getDai();
        uint256 localBalance = _dai.balanceOf(address(this));
        if (_value > localBalance) {
            _getChai().draw(
                address(this),
                _value - localBalance
            );
        }
    }
```

Listing 3.9: LoanTokenLogicDai.sol

With the existence of `_getDai()`, `_getChai()`, and `_getPot()`, the routine `_dsrWithdraw()` should not have an inconsistent implementation which includes the feature of `_getDai()`.

Recommendation Use `_getDai()` instead of `_dsrWithdraw()` to get the Dai address.

Status This issue has been addressed by re-factoring the `_dsrWithdraw()` function in this commit: 24510aa.

3.5 Incessive _dsrDeposit() Call in _mintToken()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LoanTokenLogicDai
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [3]

Description

In the LoanTokenLogicDai contract, the _mintToken() internal function implements the underlying functions of mintWithChai() and mint(). When the msg.sender sends in Chai or Dai tokens, the corresponding loan tokens would be minted.

```
255     function _mintToken(
256         address receiver,
257         uint256 depositAmount,
258         bool withChai)
259         internal
260         returns (uint256 mintAmount)
261     {
262         require(depositAmount != 0, "17");

264         _settleInterest();

266         uint256 currentPrice = _tokenPrice(_totalAssetSupply(0));
267         uint256 currentChaiPrice;
268         IERC20 inAsset;

270         if (withChai) {
271             inAsset = IERC20(address(_getChai()));
272             currentChaiPrice = chaiPrice();
273         } else {
274             inAsset = IERC20(address(_getDai()));
275         }

277         require(inAsset.transferFrom(
278             msg.sender,
279             address(this),
280             depositAmount
281         ), "18");

283         _dsrDeposit();
```

Listing 3.10: LoanTokenLogicDai.sol

As an optimization strategy, _dsrDeposit() is invoked in line 283 to save incoming Dai tokens into DSR for additional earnings. However, in the case withChai == true, there's no Dai balance increased such that the _dsrDeposit() call is not necessary.

Recommendation Call `_dsrDeposit()` only in the `withChai == false` case.

Status This issue has been addressed by calling `_dsrDeposit()` only in the `withChai == false` case in this commit: 24510aa.

3.6 Zero Amount Flash Loan

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LoanTokenLogicDai`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In bZx v2.0, the `flashBorrowToken()` function allows users to borrow some tokens, call arbitrary contracts, and return those tokens back in one transaction. While reviewing the source code, we noticed that zero amount flash loans are supported (i.e., `borrowAmount == 0`). This implementation is like a free proxy contract with no visible benefit.

```
    function flashBorrowToken(
        uint256 borrowAmount,
        address borrower,
        address target,
        string calldata signature,
        bytes calldata data)
        external
        payable
        nonReentrant
        returns (bytes memory)
    {
        _checkPause();

        _settleInterest();

        IERC20 _dai;
        if (borrowAmount != 0) {
            _dai = _dsrWithdraw(borrowAmount);
        } else {
            _dai = _getDai();
        }
```

Listing 3.11: LoanTokenLogicDai.sol

Recommendation Ensure `borrowAmount` is greater than zero.

Status This issue has been addressed by requiring `borrowAmount != 0` in this commit: 24510aa.

3.7 Confused Deputy in borrow()/marginTrade()

- ID: PVE-007
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: LoanTokenLogicStandard
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

While reviewing the the code flow of borrow()'ing from an existing loan indexed by a loanId, we noticed that the public function, i.e., LoanTokenLogicStandard::borrow(), fails to validate the borrower . This leads to a critical confused deputy issue which allows an arbitrary user to impersonate a borrower for borrowing digital assets to a given receiver.

As shown in the following code snippets, there's no sanity checks against the msg.sender and the borrower when loanId is not 0. A bad actor could simply invoke borrow() with a victim address as the borrower and a loanId which was created by that victim address for stealing assets from an existing victim's loan.

```
    function borrow(
        bytes32 loanId,                 // 0 if new loan
        uint256 withdrawAmount,
        uint256 initialLoanDuration,    // duration in seconds
        uint256 collateralTokenSent,    // if 0, loanId must be provided; any ETH sent
            must equal this value
        address collateralTokenAddress, // if address(0), this means ETH and ETH must
            be sent with the call or loanId must be provided
        address borrower,
        address receiver,
        bytes memory /*loanDataBytes*/) // arbitrary order data (for future use)
        public
        payable
        usesGasToken
        returns (uint256, uint256) // returns new principal and new collateral added to
             loan
    {
        require(withdrawAmount != 0, "6");

        _checkPause();

        require(msg.value == 0 || msg.value == collateralTokenSent, "7");
        require(collateralTokenSent != 0 || loanId != 0, "8");
        require(collateralTokenAddress != address(0) || msg.value != 0 || loanId != 0,
            "9");
```

Listing 3.12: LoanTokenLogicStandard.sol

The `marginTrade()` function has a similar issue such that the bad actor could impersonate the `trader` for trading with an existing loan.

```
    function marginTrade(
        bytes32 loanId,              // 0 if new loan
        uint256 leverageAmount,
        uint256 loanTokenSent,
        uint256 collateralTokenSent,
        address collateralTokenAddress,
        address trader,
        bytes memory loanDataBytes)     // arbitrary order data
        public
        payable
        usesGasToken
        returns (uint256, uint256) // returns new principal and new collateral added to
            trade
    {
        _checkPause();

        if (collateralTokenAddress == address(0)) {
            collateralTokenAddress = wethToken;
        }
        require(collateralTokenAddress != loanTokenAddress, "11");
```

Listing 3.13: LoanTokenLogicStandard.sol

Recommendation Validate the `msg.sender` in the beginning of `borrow()` and `marginTrade()`.

Status This issue has been addressed by validating the `msg.sender` in the beginning of `borrow()` and `marginTrade()` when `loanId != 0` in this commit: 890d476.

3.8 Business Logic Error in getLoanParamsList()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LoanTokenLogicStandard`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `LoanSettings` contract, the `getLoanParamsList()` function allows the caller to retrieve the `count` entries from `userLoanParamSets` starting at index `start`. If there is no enough entries in `userLoanParamSets`, less than `count` entries would be returned. To achieve that, the `getLoanParamsList()` function keeps an `end` index which should be computed as the smaller value between `start+count` and the length of `userLoanParamSets`.

However, the current implementation computes the `end` as the smaller value between `count` and the length of `userLoanParamSets`, which is a wrong implementation, leading to an incorrect `loanParamsList` returned. For example, when `set.values.length=5`, `start=2`, and `count=1`, the current implementation returns an empty `loanParamsList` since `end=1` and `start > end`.

```
    function getLoanParamsList(
        address owner,
        uint256 start,
        uint256 count)
        external
        view
        returns (bytes32[] memory loanParamsList)
    {
        EnumerableBytes32Set.Bytes32Set storage set = userLoanParamSets[owner];

        uint256 end = count.min256(set.values.length);
        if (end == 0 || start >= end) {
            return loanParamsList;
        }
```

Listing 3.14: LoanSettings.sol

Recommendation Compute `end` as the smaller value between `(start + count)` and `set.values.length`.

Status This issue has been addressed by fixing the `end` calculation in this commit: 6ab74ba.

3.9 Inconsistent Fee Calculation in getBorrowAmount() and getRequiredCollateral()

- ID: PVE-009
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `LoanTokenLogicStandard`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the loan token contract, the `getDepositAmountForBorrow()` viewer function allows the caller to get the `depositAmount` from the `borrowAmount`. In particular, the `getRequiredCollateral()` function in the bZx contract is invoked to calculate `collateralAmountRequired`. As shown in the following code snippets, the `fee` is added to the `collateralAmountRequired` based on the rate and the `borrowAmount`.

```
            uint256 fee = isTorqueLoan ?
                _getBorrowingFee(collateralAmountRequired) :
```

```
                _getTradingFee(collateralAmountRequired);
            if (fee != 0) {
                collateralAmountRequired = collateralAmountRequired
                    .add(fee);
            }
```

Listing 3.15: LoanOpenings:: getRequiredCollateral ()

On the other hand, the `getBorrowAmountForDeposit()` function in the loan token contract also allows the caller to get the `borrowAmount` from the `depositAmount`. As shown in the following code snippets, the `fee` is substrated from the `borrowAmount`.

```
            uint256 fee = isTorqueLoan ?
                _getBorrowingFee(collateral) :
                _getTradingFee(collateral);
            if (fee != 0) {
                collateral = collateral
                    .sub(fee);
            }

            if (loanToken == collateralToken) {
                borrowAmount = collateral
                    .mul(10**20)
                    .div(marginAmount);
            } else {
                (uint256 sourceToDestRate, uint256 sourceToDestPrecision) = IPriceFeeds(
                    priceFeeds).queryRate(
                    collateralToken,
                    loanToken
                );
                if (sourceToDestPrecision != 0) {
                    borrowAmount = collateral
                        .mul(10**20)
                        .div(marginAmount)
                        .mul(sourceToDestRate)
                        .div(sourceToDestPrecision);
                }
            }
```

Listing 3.16: LoanOpenings::getBorrowAmount()

We believe it's not a fair fee calculation. For example, let's say the fee rate is 0.3% and someone wants to deposit around 1,000 DAI for borrowing around 2 ETH. If we calculate the `depositAmount` from `borrowAmount`, 1,000 DAI could borrow $2 \times (1,000/1,003) = 1.99401795$ ETH. But, if we calculate the `borrowAmount` from `depositAmount`, $2 \times (1 - 0.3\%) = 1.994$ ETH could be borrowed. Here, we see the $1.99401795 - 1.994 = 0.00001795$ ETH difference, which is due to the inconsistent fee calculation. A fair calculation should be $depositAmount = depositAmount/(1 - 0.3\%)$.

Recommendation Fix the fee calculation on the `depositAmount` side as $depositAmount = depositAmount/(1 - fee)$.

Status This issue has been addressed by fixing the fee calculation in this commit: 0e98605.

3.10 Reentrancy Risk in withdrawAccruedInterest()

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LoanTokenLogicStandard
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

In the LoanMaintenance contract, the external function, i.e., withdrawAccruedInterest(), allows users to collect the outstanding interest. We identified a reentrancy risk such that a bad actor could redo the interest collection from LoanMaintenance contract even the interest may not be due yet.

```
function withdrawAccruedInterest(
    address loanToken)
    external
{
    // pay outstanding interest to lender
    _payInterest(
        msg.sender, // lender
        loanToken
    );
}
```

Listing 3.17: LoanMaintenance::withdrawAccruedInterest()

The reentrancy risk is in the underlying function, _payInterest(), which invokes _payInterestTransfer() (line 40) to pay the interestToken to lender. As shown in the code snippets, the time frame between block.timestamp and lenderInterestLocal.updatedTimestamp is used to calculate interestOwedNow. However, the lenderInterestLocal.updatedTimestamp is reset after the _payInterestTransfer() call, which leads to a reentrancy scenario.

```
17 function _payInterest(
18     address lender,
19     address interestToken)
20     internal
21 {
22     LenderInterest storage lenderInterestLocal = lenderInterest[lender][
           interestToken];

24     uint256 interestOwedNow = 0;
25     if (lenderInterestLocal.owedPerDay != 0 && lenderInterestLocal.updatedTimestamp
           != 0) {
26         interestOwedNow = block.timestamp
```

```
                    .sub(lenderInterestLocal.updatedTimestamp)
                    .mul(lenderInterestLocal.owedPerDay)
                    .div(86400);

                if (interestOwedNow > lenderInterestLocal.owedTotal)
                    interestOwedNow = lenderInterestLocal.owedTotal;

                if (interestOwedNow != 0) {
                    lenderInterestLocal.paidTotal = lenderInterestLocal.paidTotal
                        .add(interestOwedNow);
                    lenderInterestLocal.owedTotal = lenderInterestLocal.owedTotal
                        .sub(interestOwedNow);

                    _payInterestTransfer(
                        lender,
                        interestToken,
                        interestOwedNow
                    );
                }
            }

            lenderInterestLocal.updatedTimestamp = block.timestamp;
        }
```

Listing 3.18: InterestUser::_payInterest()

If the `interestToken` is an ERC777, the `_payInterestTransfer()` could be hijacked after the `transfer()` to re-enter the unprotected `withdrawAccruedInterest()` function. Since the `lenderInterestLocal.updatedTimestamp` is not reset yet, `interestOwedNow` would be re-calculated and `interestToken` would be sent out again.

Fortunately, the `interestToken` is not an ERC777 token such that we set the severity of this issue informational.

Recommendation Add reentrancy guard in the entry point of `withdrawAccruedInterest()` or apply the Checks-Effects-Interactions [2] pattern.

Status This issue has been addressed by resetting the `lenderInterestLocal.updatedTimestamp` before the `_payInterestTransfer()` call in this commit: 0e98605.

3.11 Unused Variables in _initializeLoan()/_closeLoan()

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LoanOpenings`, `LoanClosings`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [3]

Description

While reviewing the lifetime of a loan, we notice that the variable `pendingTradesId` that is initialized in `_initializeLoan()` when a loan is created is never used (even after the loan is closed by the `_closeLoan()` function).

```
            loanLocal = Loan({
                id: loanId,
                loanParamsId: loanParamsLocal.id,
                pendingTradesId: 0,
                active: true,
                principal: newPrincipal,
                collateral: 0, // calculated later
                startTimestamp: block.timestamp,
                endTimestamp: 0, // calculated later
                startMargin: initialMargin,
                startRate: 0, // queried later
                borrower: borrower,
                lender: lender
            });
```

Listing 3.19: LoanOpenings:: _initializeLoan ()

As shown in the following code snippets, the only use case of `pendingTradesId` is setting it to 0 when the loan is closed and removed from the `lenderLoanSets` and `borrowerLoanSets`.

```
    function _closeLoan(
        Loan storage loanLocal,
        uint256 loanCloseAmount)
        internal
        returns (uint256)
    {
        require(loanCloseAmount != 0, "nothing to close");

        if (loanCloseAmount == loanLocal.principal) {
            loanLocal.principal = 0;
            loanLocal.active = false;
            loanLocal.endTimestamp = block.timestamp;
            loanLocal.pendingTradesId = 0;
            activeLoansSet.remove(loanLocal.id);
            lenderLoanSets[loanLocal.lender].remove(loanLocal.id);
```

```
            borrowerLoanSets[loanLocal.borrower].remove(loanLocal.id);
        } else {
            loanLocal.principal = loanLocal.principal
                .sub(loanCloseAmount);
        }
    }
```

Listing 3.20: LoanClosings::_closeLoan()

Recommendation Removed the unused `pendingTradesId` variable.

3.12 Inconsistent Book-Keeping Records/Events Data in _payFeeReward()

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FeesHelper`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the `FeesHelper` contract, the `_payFeeReward()` helper function allows the caller to pay protocol tokens to `user` as rewards. However, we identified that the amount paid to the `user` could be inconsistent when compared with internal book-keeping records (i.e., `protocolTokenPaid`) as the underlying function `_withdrawProtocolToken()` could transfer less than `rewardAmount` to the `user`. Based on that, the `EarnReward()` event emitted after updating the `protocolTokenPaid` could also be inaccurate.

```
        if (rewardAmount != 0) {
            address rewardToken;
            (rewardToken, success) = _withdrawProtocolToken(
                user,
                rewardAmount
            );
            if (success) {
                protocolTokenPaid = protocolTokenPaid
                    .add(rewardAmount);

                emit EarnReward(
                    user,
                    rewardToken,
                    loanId,
                    rewardAmount
                );
            }
```

```
        }
```

Listing 3.21: FeesHelper:: _payFeeReward()

As shown in the following code snippets, the `_withdrawProtocolToken()` function mentioned earlier could transfer less than `amount` out when there's no enough protocol token balance.

```
    function _withdrawProtocolToken(
        address receiver,
        uint256 amount)
        internal
        returns (address, bool)
    {
        uint256 withdrawAmount = amount;

        uint256 balance = protocolTokenHeld;
        if (withdrawAmount > balance) {
            withdrawAmount = balance;
        }
        if (withdrawAmount == 0) {
            return (protocolTokenAddress, false);
        }

        protocolTokenHeld = balance
            .sub(withdrawAmount);

        IERC20(protocolTokenAddress).safeTransfer(
            receiver,
            withdrawAmount
        );

        return (protocolTokenAddress, true);
    }
```

Listing 3.22: ProtocolTokenUser::_withdrawProtocolToken()

Recommendation Add a return value in the `_withdrawProtocolToken()` to report the caller the exact amount of protocol token transferred.

Status This issue has been addressed by returning the `rewardAmount` in `_withdrawProtocolToken()` and refactoring the callers such as `_payFeeReward()` in this commit: 4e06df4.

3.13 Incompatibility With Deflationary Tokens in swapExternal()

- ID: PVE-013
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: SwapsExternal
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the SwapsExternal contract, the swapExternal() public function allows users to swap sourceToken to destToken through external exchange services. Before doing the swap, the swapExternal() requires the msg.sender to transfer in the sourceToken with the safeTransferFrom() handler if the caller is not paying ether. When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
        } else {
            IERC20(sourceToken).safeTransferFrom(
                msg.sender,
                address(this),
                sourceTokenAmount
            );
        }

        (destTokenAmountReceived, sourceTokenAmountUsed) = _swapsCall(
            [
                sourceToken,
                destToken,
                receiver,
                returnToSender,
                msg.sender // user
            ],
            [
                sourceTokenAmount, // minSourceTokenAmount
                sourceTokenAmount, // maxSourceTokenAmount
                requiredDestTokenAmount
            ],
```

Listing 3.23: SwapsExternal::swapExternal()

However, in the cases of deflationary tokens, as shown in the above code snippets, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate portfolio management and affects protocol-wide operation and maintenance.

Recommendation Check the `sourceToken` balance before and after the `safeTransferFrom()` call.

Status This issue has been addressed by checking the balance before and after the `safeTransferFrom()` call in this commit: 2cc224c.

3.14 Improved Arithmetic Operations

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LoanTokenLogicStandard`, `LoanOpenings`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

While reviewing the arithmetic operations in bZx v2.0, we identified some cases which could be further improved.

Case I The `mul(365)` in line 1003 could be done before `div(assetBorrow)` (line 1002) to improve the precision of `interestOwedPerDay`.


```
992     function _avgBorrowInterestRate(
993         uint256 assetBorrow)
994         internal
995         view
996         returns (uint256)
997     {
998         if (assetBorrow != 0) {
999             (uint256 interestOwedPerDay,) = _getAllInterest();
1000            return interestOwedPerDay
1001                .mul(10**20)
1002                .div(assetBorrow)
1003                .mul(365);
1004        }
1005    }
```

Listing 3.24: LoanTokenLogicStandard::_avgBorrowInterestRate()

Case II The `mul(maxDuration)` in line 1191 could be done before `div(31536000)` (line 1190) to improve the precision of `interestRate`.

```
    function _adjustValue(
        uint256 interestRate,
        uint256 maxDuration,
        uint256 marginAmount)
        internal
        pure
        returns (uint256)
    {
        return maxDuration != 0 ?
            interestRate
                .mul(10**20)
                .div(31536000) // 86400 * 365
                .mul(maxDuration)
                .div(marginAmount)
                .add(10**20) :
            10**20;
    }
```

Listing 3.25: LoanTokenLogicStandard::_adjustValue()

Case III The `mul(sourceToDestRate)` in line 210 could be done before `div(marginAmount)` (line 209) to improve the precision of `borrowAmount`.

```
197             if (loanToken == collateralToken) {
198                 borrowAmount = collateral
199                     .mul(10**20)
200                     .div(marginAmount);
201             } else {
202                 (uint256 sourceToDestRate, uint256 sourceToDestPrecision) = IPriceFeeds(
                        priceFeeds).queryRate(
203                     collateralToken,
204                     loanToken
205                 );
206                 if (sourceToDestPrecision != 0) {
207                     borrowAmount = collateral
208                         .mul(10**20)
209                         .div(marginAmount)
210                         .mul(sourceToDestRate)
211                         .div(sourceToDestPrecision);
212                 }
213             }
```

Listing 3.26: LoanOpenings::getBorrowAmount()

Case IV While calculating the `lendingFee` in `_payInterestTransfer()`, we normally want to round the fee up instead of rounding it down. Based on that, we could use `divCeil()` to replace the `div(10**20)` in line 205 to round up the `lendingFee` to the nearest $N \times 10^{20}$

```
197     function _payInterestTransfer(
198         address lender,
199         address interestToken,
200         uint256 interestOwedNow)
201         internal
```

```
    {
        uint256 lendingFee = interestOwedNow
            .mul(lendingFeePercent)
            .div(10**20);
```

Listing 3.27: InterestUser :: _payInterestTransfer()

Recommendation Do multiplications before devisions to improve the precision of the arithmetic operations. Also, use `divCeil()` to round-up the fee.

Status This issue has been addressed in this commit: 2cc224c.

3.15 Business Error in _updateCheckpoints

- ID: PVE-015
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LoanTokenLogicStandard`, `LoanOpenings`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the loan token contract, whenever some tokens are minted or burned, the `_updateCheckpoints()` function is invoked to update the stats to reflect the balance changes. However, we identified a business logic error while updating the `_currentProfit` storage indexed by the hash of `iToken_ProfitSoFar`.

```
    function _updateCheckpoints(
        address _user,
        uint256 _oldBalance,
        uint256 _newBalance,
        uint256 _currentPrice)
        internal
    {
        // keccak256("iToken_ProfitSoFar")
        bytes32 slot = keccak256(
            abi.encodePacked(_user, uint256(0
                x37aa2b7d583612f016e4a4de4292cb015139b3d7762663d06a53964912ea2fb6))
        );

        uint256 _currentProfit;
        if (_oldBalance != 0 && _newBalance != 0) {
            _currentProfit = _profitOf(
                slot,
                _oldBalance,
                _currentPrice,
                checkpointPrices_[_user]
```

```
354                 );
355             } else if (_newBalance == 0) {
356                 _currentPrice = 0;
357             }

359             assembly {
360                 sstore(slot, _currentProfit)
361             }
```

Listing 3.28: LoanTokenLogicStandard::_updateCheckpoints()

As shown in the above code snippets, the `_currentPrice` is re-calculated in line 349 only when `_oldBalance != 0 && _newBalance != 0`. Meanwhile, we don't need to `sstore` the `_currentPrice` when it is not re-calculated. In addition, since the local variable `_currentPrice` is 0 as it's not initialized, the `sstore` in line 360 typically clear the storage if the `_oldBalance == 0` or `_newBalance == 0`.

Recommendation Store `_currentPrice` into the storage only when it's updated.

```
    function _updateCheckpoints(
        address _user,
        uint256 _oldBalance,
        uint256 _newBalance,
        uint256 _currentPrice)
        internal
    {
        // keccak256("iToken_ProfitSoFar")
        bytes32 slot = keccak256(
            abi.encodePacked(_user, uint256(0
                x37aa2b7d583612f016e4a4de4292cb015139b3d7762663d06a53964912ea2fb6))
        );

        uint256 _currentProfit;
        if (_oldBalance != 0 && _newBalance != 0) {
            _currentProfit = _profitOf(
                slot,
                _oldBalance,
                _currentPrice,
                checkpointPrices_[_user]
            );

            assembly {
                sstore(slot, _currentProfit)
            }
        }

        if (_newBalance == 0) {
            _currentPrice = 0;
        }
```

Listing 3.29: LoanTokenLogicStandard::_updateCheckpoints()

Status This issue has been addressed by refactoring the `_updateCheckpoints()` function in this commit: 2cc224c.

3.16 Business Logic Error in queryReturn()

- ID: PVE-016
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PriceFeeds`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

As indicated in the comments (line 66), the `queryReturn()` function should return 0 during a pause (i.e., `globalPricingPaused == true`). However, the underlying `_queryRate()` may revert, which makes the implementation not consistent to the design.

```
66     //// NOTE: This function returns 0 during a pause, rather than a revert. Ensure
           calling contracts handle correctly. ///
67     function queryReturn(
68         address sourceToken,
69         address destToken,
70         uint256 sourceAmount)
71         public
72         view
73         returns (uint256 destAmount)
74     {
75         (uint256 rate, uint256 precision) = _queryRate(
76             sourceToken,
77             destToken
78         );

80         destAmount = sourceAmount
81             .mul(rate)
82             .div(precision);
83     }
```

Listing 3.30: PriceFeeds :: queryReturn()

As shown in the following code snippets, the first line in `_queryRate()` reverts during a pause (line 344), which makes its caller, `queryReturn()`, reverts during a pause as well.

```
337     function _queryRate(
338         address sourceToken,
339         address destToken)
340         internal
341         view
342         returns (uint256 rate, uint256 precision)
```

```
    {
        require(!globalPricingPaused, "pricing is paused");
```

Listing 3.31: PriceFeeds::_queryRate()

Recommendation Check `globalPricingPaused` in `queryReturn()` and return 0 when `globalPricingPaused == true`.

Status This issue has been addressed in this commit: 2cc224c.

3.17 Other Suggestions

Last but not least, 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 need to kick in at the very moment when the contracts are being deployed in mainnet.

4 | Conclusion

In this audit, we thoroughly analyzed the bZx v2.0 design and implementation. The system presents a unique offering of lending and margin trading platform, and we are impressed by the design and implementation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [11, 12, 13, 14, 16].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [17] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] ethereum. Security Considerations. https://solidity.readthedocs.io/en/v0.6.4/security-considerations.html#use-the-checks-effects-interactions-pattern.

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

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

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

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

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

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

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

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

[11] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[12] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[13] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[14] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[16] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[17] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.