Document Properties

<table>
<thead>
<tr>
<th>Client</th>
<th>bZeroX, LLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Smart Contract Audit Report</td>
</tr>
<tr>
<td>Target</td>
<td>bZx v2.0</td>
</tr>
<tr>
<td>Version</td>
<td>1.0-rc1</td>
</tr>
<tr>
<td>Author</td>
<td>Chiachih Wu</td>
</tr>
<tr>
<td>Auditors</td>
<td>Xuxian Jiang, Chiachih Wu, Huaguo Shi, Jeff Liu</td>
</tr>
<tr>
<td>Reviewed by</td>
<td>Jeff Liu</td>
</tr>
<tr>
<td>Approved by</td>
<td>Xuxian Jiang</td>
</tr>
<tr>
<td>Classification</td>
<td>Confidential</td>
</tr>
</tbody>
</table>

Version Info

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0-rc1</td>
<td>Sep. 1, 2020</td>
<td>Chiachih Wu</td>
<td>Release Candidate #1</td>
</tr>
</tbody>
</table>

Contact

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Shuxiao Wang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone</td>
<td>+86 173 6454 5338</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:contact@peckshield.com">contact@peckshield.com</a></td>
</tr>
</tbody>
</table>
# Contents

1 **Introduction** ........................................... 5  
   1.1 About bZx v2.0 ........................................ 5  
   1.2 About PeckShield ........................................ 6  
   1.3 Methodology ........................................... 6  
   1.4 Disclaimer ............................................. 8  

2 **Findings** ............................................. 10  
   2.1 Summary ............................................... 10  
   2.2 Key Findings ........................................... 11  

3 **Detailed Results** .................................... 12  
   3.1 Business Logic Error in `_burnToken()` .................. 12  
   3.2 Denial-of-Service Risk in borrow() ..................... 13  
   3.3 Business Logic Error in marginTrade() ................ 15  
   3.4 Incompatible `_dsrWithdraw()` Return Value ......... 17  
   3.5 Incessive `_dsrDeposit()` Call in `_mintToken()` .... 18  
   3.6 Zero Amount Flash Loan ................................ 19  
   3.7 Confused Deputy in borrow()/marginTrade() ........... 20  
   3.8 Business Logic Error in getLoanParamsList() .......... 21  
   3.9 Inconsistent Fee Calculation in getBorrowAmount() and getRequiredCollateral() ............................. 22  
   3.10 Reentrancy Risk in withdrawAccruedInterest() ....... 24  
   3.11 Unused Variables in `_initializeLoan()`/_closeLoan() ... 26  
   3.12 Inconsistent Book-Keeping Records/Events Data in `_payFeeReward()` .................. 27  
   3.13 Incompatibility With Deflationary Tokens in swapExternal() ........................................ 29  
   3.14 Improved Arithmetic Operations ......................... 30  
   3.15 Business Error in `_updateCheckpoints` ............... 32  
   3.16 Business Logic Error in queryReturn() ................ 34  
   3.17 Other Suggestions ..................................... 35  

4 **Conclusion** ............................................ 36
5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

5.1.2 Ownership Takeover

5.1.3 Redundant Fallback Function

5.1.4 Overflows & Underflows

5.1.5 Reentrancy

5.1.6 Money-Giving Bug

5.1.7 Blackhole

5.1.8 Unauthorized Self-Destruct

5.1.9 Revert DoS

5.1.10 Unchecked External Call

5.1.11 Gasless Send

5.1.12 Send Instead Of Transfer

5.1.13 Costly Loop

5.1.14 (Unsafe) Use Of Untrusted Libraries

5.1.15 (Unsafe) Use Of Predictable Variables

5.1.16 Transaction Ordering Dependence

5.1.17 Deprecated Uses

5.2 Semantic Consistency Checks

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

5.3.2 Make Visibility Level Explicit

5.3.3 Make Type Inference Explicit

5.3.4 Adhere To Function Declaration Strictly

References
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>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issuer</td>
<td>bZeroX, LLC</td>
</tr>
<tr>
<td>Website</td>
<td><a href="https://bzx.network/">https://bzx.network/</a></td>
</tr>
<tr>
<td>Type</td>
<td>Ethereum Smart Contract</td>
</tr>
<tr>
<td>Platform</td>
<td>Solidity</td>
</tr>
<tr>
<td>Audit Method</td>
<td>Whitebox</td>
</tr>
<tr>
<td>Latest Audit Report</td>
<td>Sep. 1, 2020</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Critical</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration</td>
<td>Weaknesses in this category are typically introduced during the configuration of the software.</td>
</tr>
<tr>
<td>Data Processing Issues</td>
<td>Weaknesses in this category are typically found in functionality that processes data.</td>
</tr>
<tr>
<td>Numeric Errors</td>
<td>Weaknesses in this category are related to improper calculation or conversion of numbers.</td>
</tr>
<tr>
<td>Security Features</td>
<td>Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)</td>
</tr>
<tr>
<td>Time and State</td>
<td>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.</td>
</tr>
<tr>
<td>Error Conditions, Return Values, Status Codes</td>
<td>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.</td>
</tr>
<tr>
<td>Resource Management</td>
<td>Weaknesses in this category are related to improper management of system resources.</td>
</tr>
<tr>
<td>Behavioral Issues</td>
<td>Weaknesses in this category are related to unexpected behaviors from code that an application uses.</td>
</tr>
<tr>
<td>Business Logics</td>
<td>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.</td>
</tr>
<tr>
<td>Initialization and Cleanup</td>
<td>Weaknesses in this category occur in behaviors that are used for initialization and breakdown.</td>
</tr>
<tr>
<td>Arguments and Parameters</td>
<td>Weaknesses in this category are related to improper use of arguments or parameters within function calls.</td>
</tr>
<tr>
<td>Expression Issues</td>
<td>Weaknesses in this category are related to incorrectly written expressions within code.</td>
</tr>
<tr>
<td>Coding Practices</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Severity</th>
<th># of Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>1</td>
</tr>
<tr>
<td>High</td>
<td>2</td>
</tr>
<tr>
<td>Medium</td>
<td>2</td>
</tr>
<tr>
<td>Low</td>
<td>4</td>
</tr>
<tr>
<td>Informational</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>16</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Severity</th>
<th>Title</th>
<th>Category</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVE-001</td>
<td>High</td>
<td><strong>Business Logic Error in burnToken()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-002</td>
<td>Low</td>
<td><strong>Denial-of-Service Risk in borrow()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-003</td>
<td>High</td>
<td><strong>Business Logic Error in marginTrade()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-004</td>
<td>Info.</td>
<td><strong>Incompatible dsrWithdraw() Return Value</strong></td>
<td>Coding Practices</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-005</td>
<td>Info.</td>
<td><strong>Incessive dsrDeposit() Call in mintToken()</strong></td>
<td>Coding Practices</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-006</td>
<td>Info.</td>
<td><strong>Zero Amount Flash Loan</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-007</td>
<td>Critical</td>
<td><strong>Confused Deputy in borrow() / marginTrade()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-008</td>
<td>Medium</td>
<td><strong>Business Logic Error in getLoanParamsList()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-009</td>
<td>Medium</td>
<td><strong>Inconsistent Fee Calculation in getBorrowAmount() and getRequiredCollateral()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-010</td>
<td>Info.</td>
<td><strong>Reentrancy Risk in withdrawAccruedInterest()</strong></td>
<td>Security Features</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-011</td>
<td>Info.</td>
<td><strong>Unused Variables in initializeLoan() / closeLoan()</strong></td>
<td>Coding Practices</td>
<td>Confirmed</td>
</tr>
<tr>
<td>PVE-012</td>
<td>Low</td>
<td><strong>Inconsistent Book-Keeping Records/Events Data in payFeeReward()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-013</td>
<td>Low</td>
<td><strong>Incompatibility With Deflationary Tokens in swapExternal()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-014</td>
<td>Info.</td>
<td><strong>Improved Arithmetic Operations</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-015</td>
<td>Info.</td>
<td><strong>Business Error in updateCheckpoints</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
<tr>
<td>PVE-016</td>
<td>Low</td>
<td><strong>Business Logic Error in queryReturn()</strong></td>
<td>Business Logics</td>
<td>Fixed</td>
</tr>
</tbody>
</table>

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.

Listing 3.1: LoanTokenLogicDai.sol

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

```solidity
if (toChai) {
    _dsrDeposit();

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

    success = _chai.move(
        address(this),
        msg.sender,
        amountPaid
    );

    // get Chai amount withdrawn
    amountPaid = chaiBalance
        .sub(_chai.balanceOf(address(this)));
} else {
    success = _dsrWithdraw(amountPaid).transfer(
        receiver,
        amountPaid
    );
    _dsrDeposit();
}
```

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 payable
58     nonReentrant
59     returns (uint256 newPrincipal , uint256 newCollateral )
60     {
61         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 ( loanId ,
178     withdrawAmount ,
179     2 * 10**18 , // leverageAmount (translates to 150% margin for a Torque loan)
180     collateralTokenAddress ,
181     sentAddresses ,
182     sentAmounts ,
183     "" // loanDataBytes
184 );
```

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.

```
843 uint256 msgValue ;
844 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.

```solidity
if (loanDataBytes.length == 0) {
    (destTokenAmountReceived, sourceTokenAmountUsed) = _swapsCall_internal(
        addr,
        vals
    );
} else {
    /*
    // keccak256("Swaps_SwapsImplZeroX")
    address swapsImplZeroX;
    assembly {
        swapsImplZeroX := sload(0
        x129a6cb350d136ca8d0881f83a9141afd5dc8b3c99057f06df01ab75943df952)
    }
    */
    // revert(string(loanDataBytes));
    /*
    vaultWithdraw(
        addr[0], // sourceToken
        address(zeroXConnector),
        sourceTokenAmount
    );
    (destTokenAmountReceived, sourceTokenAmountUsed) = zeroXConnector.swap.value
    (msg.value)(
        addr[0], // sourceToken
        addr[1], // destToken
        addr[2], // receiver
        sourceTokenAmount, 0,
        loanDataBytes
    );
    */
}
```

Listing 3.7: Swaps.sol::_swapsCall()

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

```solidity
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.

```solidity
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 \texttt{msg.sender} sends in Chai or Dai tokens, the corresponding loan tokens would be minted.

```solidity
function \_mintToken(
    address receiver,
    uint256 depositAmount,
    bool withChai)
internal
returns (uint256 mintAmount)
{
    require (depositAmount != 0, "17");

    \_settleInterest();

    uint256 currentPrice = \_tokenPrice(_totalAssetSupply(0));
    uint256 currentChaiPrice;
    IERC20 inAsset;

    if (withChai) {
        inAsset = IERC20(address(_getChai()));
        currentChaiPrice = chaiPrice();
    } else {
        inAsset = IERC20(address(_getDai()));
    }

    require(inAsset.transferFrom(
        msg.sender,
        address(this),
        depositAmount
    ), "18");

    \_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.

```solidity
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 code flow of `borrow()` 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.

```solidity
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.

```solidity
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) :
```

---

Confidential
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.

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 $2 \times \frac{1,000}{1.003} = 1.99401795$ 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 − fee).

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.

```solidity
function withdrawAccruedInterest(address loanToken) public 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.

```solidity
function _payInterest(address lender, address interestToken) internal {
    // storage lenderInterestLocal = lenderInterest[lender][interestToken];

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


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).

```java
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`.

```java
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);
    }
}
```

26/43 PeckShield Audit Report #: 2020-21
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.
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.

```solidity
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);
}
```


**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 \texttt{swapExternal()}

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

Description

In the \texttt{SwapsExternal} contract, the \texttt{swapExternal()} public function allows users to swap \texttt{sourceToken} to \texttt{destToken} through external exchange services. Before doing the swap, the \texttt{swapExternal()} requires the \texttt{msg.sender} to transfer in the \texttt{sourceToken} with the \texttt{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.

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`.

```solidity
function _avgBorrowInterestRate(
    uint256 assetBorrow
) internal view
returns (uint256)
{
    if (assetBorrow != 0) {
        (uint256 interestOwedPerDay,) = _getAllInterest();
        return interestOwedPerDay
        .mul(10**20)
        .div(assetBorrow)
        .mul(365);
    }
}
```

**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`.

```solidity
function _getMaxDuration(
    LoanInterface loan
) internal view
returns (uint256)
{
    return _getMaxDurationInternal(loan)
        .mul(365)
        .div(31536000);
}
```

**Listing 3.25:** `LoanTokenLogicStandard::_getMaxDuration()`
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.

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 \(N \times 10^{20}\)
Listing 3.27: InterestUser :: _payInterestTransfer()

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

Status  This issue has been addressed in this commit: \texttt{2cc224c}.

3.15 Business Error in \_updateCheckpoints

- ID: PVE-015
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

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 \texttt{iToken\_ProfitSoFar}.
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.

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 Logsics [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.

```
// // NOTE: This function returns 0 during a pause, rather than a revert. Ensure
// calling contracts handle correctly. ///

function queryReturn(
    address sourceToken,
    address destToken,
    uint256 sourceAmount)
public
view
returns (uint256 destAmount)
{
    (uint256 rate, uint256 precision) = _queryRate(
        sourceToken, 
        destToken
    );

destAmount = sourceAmount
    .mul(rate)
    .div(precision);
}
```

**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.

```
function _queryRate(
    address sourceToken,
    address destToken)
internal
view
returns (uint256 rate, uint256 precision)
```
Listing 3.31: PriceFeeds::_queryRate()

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

**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 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


