



SMART CONTRACT AUDIT REPORT

for

O3 Interchange



Prepared By: Patrick Lou

PeckShield
March 15, 2022

Document Properties

Client	O3 Labs
Title	Smart Contract Audit Report
Target	O3 Interchange
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 15, 2022	Xiaotao Wu	Final Release
1.0-rc	March 13, 2022	Xiaotao Wu	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About O3 Interchange	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Incompatibility with Deflationary/Rebasing Tokens	11
3.2	Meaningful Events For Important State Changes	12
3.3	Suggested Fine-Grained Risk Control Of Transfer Volume	14
3.4	Trust Issue of Admin Keys	16
4	Conclusion	21
	References	22

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the O3 Interchange feature, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About O3 Interchange

O3 Interchange is the version 2 of the O3 Swap protocol. It is a cross-chain DEX and provides services in bridging the same token across different chains, as well as swapping heterogeneous or different digital assets. In addition to aggregating DEXs on the source chain as the v1 protocol did, O3 Interchange also aggregates DEXs on the destination chain. This means users will have more choices for their destination assets. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of O3 Interchange

Item	Description
Name	O3 Labs
Website	https://o3swap.com/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 15, 2022

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

- <https://github.com/O3Labs/o3swap-v2-core.git> (8b72fad)

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

- <https://github.com/O3Labs/o3swap-v2-core.git> (341ff97)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

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

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

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

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

Table 1.3: The Full Audit Checklist

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

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

1.4 Disclaimer

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




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

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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the 03 `Interchange` smart contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	1	
Informational	2	
Total	4	

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 medium-severity vulnerability, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key O3 Interchange Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Resolved
PVE-002	Informational	Meaningful Events For Important State Changes	Coding Practices	Resolved
PVE-003	Informational	Suggested Fine-Grained Risk Control Of Transfer Volume	Security Features	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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

3 | Detailed Results

3.1 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: PToken
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In 03 Interchange, the PToken contract provides an external `deposit()` function for users to transfer the supported assets (e.g., `_tokenUnderlying`) to the PToken contract and mint the corresponding amount of PToken to the users. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the PToken contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the contract's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the `deposit()` routine that is used to transfer `_tokenUnderlying` to the PToken contract.

```

61 // deposit input amount is the original token amount
62 // e.g. USDT decimals is 6 , pUSDT decimals is 18
63 // when deposit 1$ USDT , amount is 10**6 , and you'll receive 10**18 pUSDT
64 function deposit(address to, uint256 amount) external onlyDepositWithdrawEnabled {
65     require(amount != 0, "deposit amount cannot be zero");
66
67     IERC20(_tokenUnderlying).safeTransferFrom(_msgSender(), address(this), amount);
68     _mint(to, _precisionConversion(false, amount));
69
70     emit Deposit(to, amount);
71 }

```

Listing 3.1: PToken::deposit()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every [transfer](#)

() or `transferFrom()`. (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `buyFdepositund()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into 03 Interchange for trading. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status This issue has been fixed in the following commit: 916d0bb.

3.2 Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [2]

Description

In Ethereum, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `PToken` contract as an example. While examining the events that reflect the `PToken` dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the `setAuthorizedCaller()/removeAuthorizedCaller()/enableDepositWithdraw()/disableDepositWithdraw()/setWithdrawFee()` are being called, there are no

corresponding events being emitted to reflect the occurrence of `setAuthorizedCaller()/removeAuthorizedCaller()/enableDepositWithdraw()/disableDepositWithdraw()/setWithdrawFee()`.

```

95     function setAuthorizedCaller(address caller) external onlyOwner {
96         _authorizedCaller[caller] = true;
97     }

99     function removeAuthorizedCaller(address caller) external onlyOwner {
100         _authorizedCaller[caller] = false;
101     }

```

Listing 3.2: PToken::setAuthorizedCaller()/removeAuthorizedCaller()

```

103     function enableDepositWithdraw() external onlyOwner {
104         _depositWithdrawEnabled = true;
105     }

107     function disableDepositWithdraw() external onlyOwner {
108         _depositWithdrawEnabled = false;
109     }

```

Listing 3.3: PToken::enableDepositWithdraw()/disableDepositWithdraw()

```

111     function setWithdrawFee(uint256 withdrawFeeRate_, address feeCollector_) external
        onlyOwner {
112         require(withdrawFeeRate_ <= FEE_DENOMINATOR, "new withdraw fee exceeds maximum")
            ;

114         _withdrawFeeRate = withdrawFeeRate_;
115         _feeCollector = feeCollector_;
116     }

```

Listing 3.4: PToken::setWithdrawFee()

Note a number of routines in the 03 Interchange contracts can be similarly improved, including `CallProxy::setWETH()/setBridge()/enableExternalCall()/disableExternalCall()` and `Wrapper::setBridgeContract()/setFeeCollector()/setWETHAddress()`.

Recommendation Properly emit the related event when the above-mentioned functions are being invoked.

Status This issue has been fixed in the following commit: 971dd79.

3.3 Suggested Fine-Grained Risk Control Of Transfer Volume

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Bridge
- Category: Security Features [5]
- CWE subcategory: CWE-654 [3]

Description

According to the 03 Interchange design, the PToken contract will likely accumulate a huge amount of assets with the increased popularity of cross-chain transactions. While examining the implementation of the Bridge, we notice there is no risk control based on the requested transfer amount, including but not limited to daily transfer volume restriction and per-transaction transfer volume restriction. This is reasonable under the assumption that the protocol will always work well without any vulnerability and the admin keys are always properly managed. In the following, we take the Bridge::bridgeOut()/bridgeIn() routines to elaborate our suggestion.

Specifically, we show below the related code snippet of the Bridge contract. According to the 03 Interchange design, when the bridgeOut() function is called on the source chain, the bridgeIn() function on the destination chain will be called subsequently to transfer a certain amount of assets to the recipient, in order to reach the cross-chain transfer purpose. Considering the unlikely situation where the admin keys may be hijacked or leaked, all the assets locked up in the PToken contract will be stolen. To mitigate, we suggest to add fine-grained risk controls based on the requested transfer volume. A guarded launch process is also highly recommended.

```

156     function bridgeOut(
157         address fromAssetHash,
158         uint64 toChainId,
159         bytes memory toAddress,
160         uint256 amount,
161         bytes memory callData
162     ) public override returns(bool) {
163         require(amount != 0, "amount cannot be zero!");

165         // check if bridge fee is required
166         uint256 bridgeFee = 0;
167         if (bridgeFeeRate == 0 & bridgeFeeCollector == address(0)) {
168             // no bridge fee
169         } else {
170             bridgeFee = amount.mul(bridgeFeeRate).div(FEE_DENOMINATOR);
171             amount = amount.sub(bridgeFee);
172             require(_chargeFee(fromAssetHash, _msgSender(), bridgeFeeCollector,
173                             bridgeFee), "charge fee failed!");
173         }

```

```

175         require(_burnFrom(fromAssetHash, _msgSender(), amount), "transfer and burn asset
            from fromAddress to bridge contract failed!");
177     return _bridgeOut(fromAssetHash, toChainId, toAddress, amount, callData);
178 }

```

Listing 3.5: Bridge::bridgeOut()

```

211     function bridgeIn(bytes memory argsBs, bytes memory fromContractAddr, uint64
        fromChainId) onlyManagerContract public returns (bool) {
212         TxArgs memory args = _deserializeTxArgs(argsBs);

214         require(fromContractAddr.length != 0, "from proxy contract address cannot be
            empty");
215         require(Utils.equalStorage(bridgeHashMap[fromChainId], fromContractAddr), "From
            Proxy contract address error!");

217         require(args.toAssetHash.length != 0, "toAssetHash cannot be empty");
218         address toAssetHash = Utils.bytesToAddress(args.toAssetHash);

220         require(args.toAddress.length != 0, "toAddress cannot be empty");
221         address toAddress = Utils.bytesToAddress(args.toAddress);

223         if (args.callData.length == 0 & callProxy == address(0)) {
224             require(_mintTo(toAssetHash, toAddress, args.amount), "mint ptoken to user
                failed");
225         } else {
226             require(_mintTo(toAssetHash, callProxy, args.amount), "mint ptoken to
                callProxy failed");
227             require(ICallProxy(callProxy).proxyCall(toAssetHash, toAddress, args.amount,
                args.callData), "execute callData via callProxy failed");
228         }

230         emit UnlockEvent(toAssetHash, toAddress, args.amount);

232         return true;
233     }

```

Listing 3.6: Bridge::bridgeIn()

Recommendation We suggest to add fine-grained risk controls, including but not limited to, daily transfer volume restriction and per-transaction transfer volume restriction.

Status This issue has been confirmed. The team confirms that every transaction with a large token value will be monitored by Poly Network's relay system and security partners.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the 03 Interchange protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., mint/burn PToken, enable/disable deposit and withdraw functions of the the PToken contract, set/remove authorized caller for the PToken contract, pause/unpause the Wrapper contract, and set the key parameters, etc.). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

52     function mint(address to, uint256 amount) external onlyAuthorizedCaller {
53         require(amount != 0, "ERC20: zero mint amount");
54         _mint(to, amount);
55     }

57     function burn(uint256 amount) external onlyAuthorizedCaller {
58         _burn(_msgSender(), amount);
59     }

```

Listing 3.7: PToken::mint()/burn()

```

95     function setAuthorizedCaller(address caller) external onlyOwner {
96         _authorizedCaller[caller] = true;
97     }

99     function removeAuthorizedCaller(address caller) external onlyOwner {
100         _authorizedCaller[caller] = false;
101     }

```

Listing 3.8: PToken::setAuthorizedCaller()/removeAuthorizedCaller()

```

103    function enableDepositWithdraw() external onlyOwner {
104        _depositWithdrawEnabled = true;
105    }

107    function disableDepositWithdraw() external onlyOwner {
108        _depositWithdrawEnabled = false;
109    }

```

Listing 3.9: PToken::enableDepositWithdraw()/disableDepositWithdraw()


```

111     function setWithdrawFee(uint256 withdrawFeeRate_, address feeCollector_) external
        onlyOwner {
112         require(withdrawFeeRate_ <= FEE_DENOMINATOR, "new withdraw fee exceeds maximum")
            ;
114         _withdrawFeeRate = withdrawFeeRate_;
115         _feeCollector = feeCollector_;
116     }

```

Listing 3.10: PToken::setWithdrawFee()

```

39     function pause() external onlyOwner {
40         _pause();
41     }

43     function unpause() external onlyOwner {
44         _unpause();
45     }

47     function setBridgeContract(address _bridge) public onlyOwner {
48         bridge = _bridge;
49     }

51     function setFeeCollector(address _feeCollector) public onlyOwner {
52         feeCollector = _feeCollector;
53     }

55     function setWETHAddress(address _weth) public onlyOwner {
56         wethAddress = _weth;
57     }

```

Listing 3.11: Wrapper::pause()/unpause()/setBridgeContract()/setFeeCollector()/setWETHAddress()

```

64     function setBridgeFee(uint256 _rate, address _feeCollector) public onlyOwner {
65         bridgeFeeRate = _rate;
66         bridgeFeeCollector = _feeCollector;
67         emit setBridgeFeeEvent(_rate, _feeCollector);
68     }

70     function setCallProxy(address _callProxy) onlyOwner public {
71         callProxy = _callProxy;
72         emit SetCallProxyEvent(_callProxy);
73     }

75     function setManagerProxy(address ethCCMProxyAddr) onlyOwner public {
76         managerProxyContract = ethCCMProxyAddr;
77         emit SetManagerProxyEvent(managerProxyContract);
78     }

80     function bindBridge(uint64 toChainId, bytes memory targetBridge) onlyOwner public
        returns (bool) {
81         bridgeHashMap[toChainId] = targetBridge;

```

```

82     emit BindBridgeEvent(toChainId, targetBridge);
83     return true;
84 }

86 function bindAssetHash(address fromAssetHash, uint64 toChainId, bytes memory
    toAssetHash) onlyOwner public returns (bool) {
87     assetHashMap[fromAssetHash][toChainId] = toAssetHash;
88     emit BindAssetEvent(fromAssetHash, toChainId, toAssetHash);
89     return true;
90 }

```

Listing 3.12: Bridge::setBridgeFee()/setCallProxy()/setManagerProxy()/bindBridge()/bindAssetHash()

```

92 function bindBridgeBatch(uint64[] memory toChainIds, bytes[] memory
    targetBridgeHashes) onlyOwner public returns (bool) {
93     require(toChainIds.length == targetBridgeHashes.length, "Inconsistent parameter
        lengths");
94     for (uint i=0; i<toChainIds.length; i++) {
95         bridgeHashMap[toChainIds[i]] = targetBridgeHashes[i];
96         emit BindBridgeEvent(toChainIds[i], targetBridgeHashes[i]);
97     }
98     return true;
99 }

101 function bindAssetHashBatch(address[] memory fromAssetHashes, uint64[] memory
    toChainIds, bytes[] memory toAssetHashes) onlyOwner public returns (bool) {
102     require(toChainIds.length == fromAssetHashes.length, "Inconsistent parameter
        lengths");
103     require(toChainIds.length == toAssetHashes.length, "Inconsistent parameter
        lengths");
104     for (uint i=0; i<toChainIds.length; i++) {
105         assetHashMap[fromAssetHashes[i]][toChainIds[i]] = toAssetHashes[i];
106         emit BindAssetEvent(fromAssetHashes[i], toChainIds[i], toAssetHashes[i]);
107     }
108     return true;
109 }

```

Listing 3.13: Bridge::bindBridgeBatch()/bindAssetHashBatch()

```

27 function setWETH(address _wethAddress) public onlyOwner {
28     wethAddress = _wethAddress;
29 }

31 function setBridge(address _bridgeAddress) public onlyOwner {
32     bridgeAddress = _bridgeAddress;
33 }

35 function enableExternalCall() public onlyOwner {
36     externalCallEnabled = true;
37 }

39 function disableExternalCall() public onlyOwner {

```

```

40     externalCallEnabled = false;
41 }

```

Listing 3.14: CallProxy::setWETH()/setBridge()/enableExternalCall()/disableExternalCall()

```

617 function applySwapFee(uint256 newSwapFee) external onlyOwner {
618     require(newSwapFee <= MAX_SWAP_FEE, "03SwapPool: swap fee exceeds maximum");
619     swapFee = newSwapFee;

621     emit NewSwapFee(newSwapFee);
622 }

624 function applyAdminFee(uint256 newAdminFee) external onlyOwner {
625     require(newAdminFee <= MAX_ADMIN_FEE, "03SwapPool: admin fee exceeds maximum");
626     adminFee = newAdminFee;

628     emit NewAdminFee(newAdminFee);
629 }

631 function withdrawAdminFee(address receiver) external onlyOwner {
632     for (uint256 i = 0; i < coins.length; i++) {
633         IERC20 token = coins[i];
634         uint256 balance = token.balanceOf(address(this)) - balances[i];
635         if (balance > 0) {
636             token.safeTransfer(receiver, balance);
637         }
638     }
639 }

641 function rampA(uint256 _futureA, uint256 _futureTime) external onlyOwner {
642     require(block.timestamp >= initialATime + MIN_RAMP_TIME, "03SwapPool: at least 1
        day before new ramp");
643     require(_futureTime >= block.timestamp + MIN_RAMP_TIME, "03SwapPool:
        insufficient ramp time");
644     require(_futureA > 0 && _futureA < MAX_A, "03SwapPool: futureA must in range (0,
        MAX_A)");

646     uint256 initialAPrecise = _getAPrecise();
647     uint256 futureAPrecise = _futureA * A_PRECISION;

649     if (futureAPrecise < initialAPrecise) {
650         require(futureAPrecise * MAX_A_CHANGE >= initialAPrecise, "03SwapPool:
            futureA too small");
651     } else {
652         require(futureAPrecise <= initialAPrecise * MAX_A_CHANGE, "03SwapPool:
            futureA too large");
653     }

655     initialA = initialAPrecise;
656     futureA = futureAPrecise;
657     initialATime = block.timestamp;
658     futureATime = _futureTime;

```

```

660     emit RampA(initialAPrecise, futureAPrecise, block.timestamp, _futureTime);
661 }

663 function stopRampA() external onlyOwner {
664     require(futureATime > block.timestamp, "03SwapPool: ramp already stopped");

666     uint256 currentA = _getAPrecise();

668     initialA = currentA;
669     futureA = currentA;
670     initialATime = block.timestamp;
671     futureATime = block.timestamp;

673     emit StopRampA(currentA, block.timestamp);
674 }

```

Listing 3.15: Pool::applySwapFee()/applyAdminFee()/withdrawAdminFee()/rampA()/stopRampA()

If the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. The team confirms that the `mint()/burn()` functions will only open to whitelisted contracts. Currently only the `Bridge` contract address (only one per chain) will be added into the whitelist.

4 | Conclusion

In this audit, we have analyzed the 03 Interchange design and implementation. 03 Interchange is the version 2 of the 03 Swap protocol. It is a cross-chain DEX and provides services in bridging the same token across different chains, as well as swapping heterogeneous or different digital assets. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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



References

- [1] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [3] MITRE. CWE-654: Reliance on a Single Factor in a Security Decision. <https://cwe.mitre.org/data/definitions/654.html>.
- [4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [5] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [6] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [7] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [8] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

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

