SMART CONTRACT AUDIT REPORT

for

Kinetex xSwap

Prepared By: Xiaomi Huang

PeckShield

March 3, 2023

Document Properties

Client	Kinetex
Title	Smart Contract Audit Report
Target	Kinetex xSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 3, 2023	Xuxian Jiang	Final Release
1.0-rc	February 26, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Kinetex xSwap` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Kinetex xSwap

`Kinetex xSwap` protocol is designed to be an aggregation protocol that allows users to perform cross-chain swaps without a need from them to do actions manually in the non-initial chain. The swap data is validated and signed by the user. The signature allows to call swap operation steps by an arbitrary executor securely. The protocol also allows manual call of a swap step by the user with no signature providing needed. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Kinetex xSwap

Item	Description
Name	Kinetex
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	March 3, 2023

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

- https://github.com/KinetexNetwork/xswap-contracts.git (3d75621)

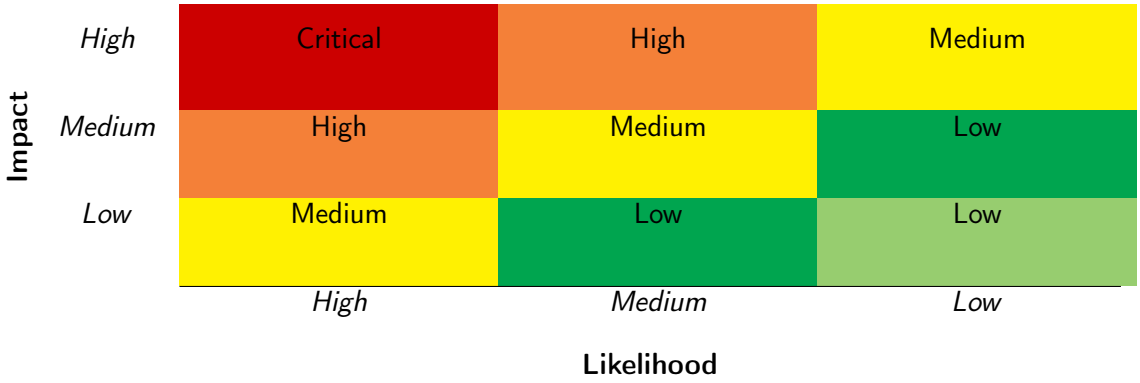
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/KinetexNetwork/xswap-contracts.git (8c8efee)

1.2 About PeckShield

PeckShield Inc. [9] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

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

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

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	2	■■
Informational	0	
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 high-severity vulnerability, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-002	High	Improved Signature Validation in SwapSignatureValidator	Business Logic	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Revisited Native Token Claim Logic in Swapper	Coding Practices	Confirmed

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

3 | Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

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

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201         // To change the approve amount you first have to reduce the addresses`
202         //  allowance to zero by calling `approve(_spender, 0)` if it is not
203         //  already 0 to mitigate the race condition described here:
204         //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.1: USDT Token Contract

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

```
38      /**
39       * @dev Deprecated. This function has issues similar to the ones found in
40       * {IERC20-approve}, and its usage is discouraged.
41       *
42       * Whenever possible, use {safeIncreaseAllowance} and
43       * {safeDecreaseAllowance} instead.
44       */
45      function safeApprove(
46          IERC20 token,
47          address spender,
48          uint256 value
49      ) internal {
50          // safeApprove should only be called when setting an initial allowance,
51          // or when resetting it to zero. To increase and decrease it, use
52          // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
53          require(
54              (value == 0)  (token.allowance(address(this), spender) == 0),
55              "SafeERC20: approve from non-zero to non-zero allowance"
56          );
57          _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
                spender, value));
58      }
```

Listing 3.2: `SafeERC20::safeApprove()`

In current implementation, if we examine the `UniswapPermitResolver::resolvePermit()` routine that is designed to resolve the user allowance. To accommodate the specific idiosyncrasy, there is a need to make use of `safeApprover()` twice: the first one resets the current allowance and the second one approves the intended amount (line 25). Note this issue may impact a number of other related routines as well.

```
21      function resolvePermit(address token_, address from_, uint256 amount_, uint256
            deadline_, bytes calldata signature_) external {
22          require(msg.sender == xSwap, "UP: caller must be xSwap");
23          uint256 nonce = uint256(keccak256(abi.encodePacked(token_, from_, amount_,
                deadline_, address(this))));
24          Permit2(PERMIT2).permitTransferFrom(PermitTransferFrom({permitted:
                TokenPermissions({token: token_, amount: amount_}), nonce: nonce, deadline:
```

```
            deadline_}), SignatureTransferDetails({to: address(this), requestedAmount:
            amount_}), from_, signature_);
25          SafeERC20.safeApprove(IERC20(token_), msg.sender, amount_);
26      }
```

Listing 3.3: `UniswapPermitResolver::resolvePermit()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in the following commit: `a01d2ec`.

3.2 Improved Signature Validation in SwapSignatureValidator

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `SwapSignatureValidator`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Kinetex xSwap` protocol makes an intensive use of user signatures to facilitate different swap operations. Naturally, there is a need to properly validate the user signature. While examining the current validation logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related `validateSwapSignature()` routine. As the name indicates, this routine ensures that the given `swap` request is indeed a legitimate one which is signed by the requesting entity. Note that the internal implementation makes use of the `ecrecover()` precompile for validation. It comes to our attention that the precompile-based validation needs to properly ensure the signer, i.e., `signer`, is not equal to `address(0)`!

```
8       function validateSwapSignature(Swap calldata swap_, bytes calldata swapSignature_)
            public pure {
9           require(swap_.steps.length > 0, "SV: swap has no steps");
10          address signer = ECDSA.recover(_hashTypedDataV4(_hashSwap(swap_), swap_.steps
                [0].chain, swap_.steps[0].swapper), swapSignature_);
11          require(signer == swap_.steps[0].account, "SV: invalid swap signature");
12      }

14      function validateStealthSwapStepSignature(SwapStep calldata swapStep_, StealthSwap
            calldata stealthSwap_, bytes calldata stealthSwapSignature_) public pure returns
             (uint256 stepIndex) {
15          address signer = ECDSA.recover(_hashTypedDataV4(_hashStealthSwap(stealthSwap_),
                stealthSwap_.chain, stealthSwap_.swapper), stealthSwapSignature_);
16          require(signer == stealthSwap_.account, "SV: invalid s-swap signature");
```

```
17          return findStealthSwapStepIndex(swapStep_, stealthSwap_);
18      }
```

Listing 3.4: SwapSignatureValidator::validateSwapSignature()/validateStealthSwapStepSignature()

Moreover, we notice it only validates the first step's account, which is insufficient to verify the legitimacy of other steps!

Recommendation Strengthen the above validateSwapSignature() routine to ensure the signer is not equal to address(0) and all swap steps need to be properly validated.

Status This issue has been fixed in the following commit: a01d2ec.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Kinetex xSwap protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and user authorization). It also has the privilege to regulate or govern the flow of assets within the protocol.

With great privilege comes great responsibility. Our analysis shows that the owner account is indeed privileged. In the following, we show representative privileged operations in the Kinetex xSwap protocol.

```
28      function addAccountToWhitelist(address account_) external whenInitialized onlyOwner
            {
29          require(_accounts.add(account_), "AW: account already included");
30          emit AccountAdded(account_);
31      }

33      function removeAccountFromWhitelist(address account_) external whenInitialized
            onlyOwner {
34          require(_accounts.remove(account_), "AW: account already excluded");
35          emit AccountRemoved(account_);
36      }
```

Listing 3.5: Various Privileged Operations in AccountWhitelist

We emphasize that the privilege assignment with various protocol contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure.

We point out that a compromised `owner` account would allow the attacker to invoke the above `drainTo` to steal funds in current protocol, which directly undermines the assumption of the `AirSwap` protocol.

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 and partially mitigated with a multi-sig account to regulate the governance privileges.

3.4 Revisited Native Token Claim Logic in Swapper

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Swapper`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, `Kinetex xSwap` is an aggregation protocol and there is a need to accommodate a variety of tokens. Specifically, when there are leftover funds, the aggregation contract needs to properly return back to the calling user. Our analysis shows the return of unused native tokens needs to be improved.

Specifically, we show below the key routine behind the performed swaps: `_performSwapStep()`. While this routine properly validates the input, transfers in user funds, performs the intended swap, and consumes the resulting funds. It does not properly return the unclaimed native coins even though the bridges have properly returned the unused ones (via `_hop()`).

```
106     function _performSwapStep(SwapStep calldata step_, Permit[] calldata permits_,
107         uint256[] calldata inAmounts_, Call calldata call_, bytes[] calldata useArgs_)
                private {
108         require(step_.deadline > block.timestamp, "SW: swap step expired");
109         require(step_.chain == block.chainid, "SW: wrong swap step chain");
110         require(step_.swapper == address(this), "SW: wrong swap step swapper");
111         require(step_.ins.length == inAmounts_.length, "SW: in amounts length mismatch")
                ;
```

```
113         _useNonce(step_.account, step_.nonce);
114         _usePermits(step_.account, permits_);
116         uint256[] memory outAmounts = _performCall(step_.account, step_.sponsor, step_.
                ins, inAmounts_, step_.outs, call_);
117         _performUses(step_.uses, useArgs_, step_.outs, outAmounts);
118     }
```

Listing 3.6: `Swapper::_performSwapStep()`

```
40      function _hop(TokenCheck calldata in_, uint256 inAmount_, uint256 chain_, address
            account_, NativeClaimer.State memory nativeClaimer_) private
            returnUnclaimedNative(nativeClaimer_) {
41          TokenHelper.transferToThis(in_.token, msg.sender, inAmount_, nativeClaimer_);
43          if (TokenHelper.isNative(in_.token)) {
44              IHyphen(hyphen).depositNative{value: inAmount_}(account_, uint64(chain_), "
                    xSwap");
45          } else {
46              TokenHelper.approveOfThis(in_.token, hyphen, inAmount_);
47              IHyphen(hyphen).depositErc20(uint64(chain_), in_.token, account_, inAmount_,
                     "xSwap");
48              TokenHelper.revokeOfThis(in_.token, hyphen);
49          }
50      }
```

Listing 3.7: `Hyphen::_hop()`

Recommendation Properly return the unspent native coins back to the user.

Status This issue has been confirmed and the team prefers to keep the related logic as-is.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Kinetex xSwap` protocol, which is designed to be an aggregation protocol that allows users to perform cross-chain swaps without a need from them to do actions manually in the non-initial chain. The swap data is validated and signed by the user. The signature allows to call swap operation steps by an arbitrary executor securely. The protocol also allows manual call of a swap step by the user with no signature providing needed. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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