SMART CONTRACT AUDIT REPORT

for

Xave Protocol

Prepared By: Xiaomi Huang

PeckShield

July 27, 2022

Document Properties

Client	Xave Finance
Title	Smart Contract Audit Report
Target	Xave
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Liu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 27, 2022	Xuxian Jiang	Final Release
1.0-rc	July 23, 2022	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 Xave 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 Xave

`Xave Finance` aims to be a one stop shop for fintechs to enable real time, cross border remittance and high yield consumer savings powered by DeFi. `Xave` does this by building an on chain stablecoin FX AMM and stablecoin focused lending market. This audit focuses on the `AMM v2` of the `Xave` protocol, essentially composed of what is called the `FXPool` - an FX accurate stablecoin pool built on top of `Balancer`'s V2 Vault. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Xave

Item	Description
Name	Xave Finance
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	July 27, 2022

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

- https://github.com/xave-finance/amm-contracts.git (3a68fce)

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

- https://github.com/xave-finance/amm-contracts.git (5adc8a1)

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

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [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 `Xave` 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	1	■
High	0	
Medium	2	■ ■
Low	1	■
Informational	1	■
Total	5	

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

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Logic in BaseToUsdAssimilator::outputRaw()	Business Logic	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Informational	Suggested immutable Use in AssimilatorFactory	Coding Practices	Resolved
PVE-004	Critical	Caller Validation in FX-Pool::onJoinPool()/onExitPool()	Business Logic	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Improper Logic in BaseToUsdAssimilator::outputRaw()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Xave` protocol supports a number of built-in `assimilator`s, which allow for the conversion among related assets. For example, both `BaseToUsdAssimilator` and `UsdcToUsdAssimilator` have defined a function `outputRaw()`, which takes a raw amount of `USDC/baseToken` and transfers it out with the numeraire value of the raw amount. Our analysis shows this function needs to be revised.

To elaborate, we use the `BaseToUsdAssimilator` contract as an example and shows below the `outputRaw()` implementation. While it is designed to return the numeraire value of the given output raw amount, the actual amount for the `transfer()` call should be the given `_amount`, not the computed numeraire amount: `baseToken.transfer(_dst, _baseTokenAmount)` (line 145). Note the same issue is applicable to the same `outputRaw()` function in the `UsdcToUsdAssimilator` contract.

```
140    function outputRaw(address _dst, uint256 _amount) external override returns (int128
           amount_) {
141        uint256 _rate = getRate();
142
143        uint256 _baseTokenAmount = (_amount * _rate) / 1e8;
144
145        bool _transferSuccess = baseToken.transfer(_dst, _baseTokenAmount);
146
147        require(_transferSuccess, 'BaseAssimilator/baseToken-transfer-failed');
148
149        amount_ = _baseTokenAmount.divu(baseDecimals);
150    }
```

Listing 3.1: BaseToUsdAssimilator::outputRaw()

Recommendation Revise the above `outputRaw()` function in the two contracts `BaseToUsdAssimilator` and `UsdcToUsdAssimilator` to use the right amount for the `transfer()` call.

Status This issue has been resolved by the following commit: `fb53c09`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

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

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

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

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

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
```

```
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `outputRawAndGetBalance()` routine in the `BaseToUsdAssimilator` contract. If the `USDT` token is supported as `baseToken`, the unsafe version of `baseToken.transfer(_dst, _baseTokenAmount)` (lines $128-129$) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
119     function outputRawAndGetBalance(address _dst, uint256 _amount)
120         external
121         override
122         returns (int128 amount_, int128 balance_)
123     {
124         uint256 _rate = getRate();
125 
126         uint256 _baseTokenAmount = ((_amount) * _rate) / 1e8;
127 
128         bool _transferSuccess = baseToken.transfer(_dst, _baseTokenAmount);
129 
130         require(_transferSuccess, 'BaseAssimilator/baseToken-transfer-failed');
131 
132         uint256 _balance = baseToken.balanceOf(address(this));
133 
134         amount_ = _baseTokenAmount.divu(baseDecimals);
135 
136         balance_ = ((_balance * _rate) / 1e8).divu(baseDecimals);
137     }
```

Listing 3.3: BaseToUsdAssimilator::outputRawAndGetBalance()

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

Status This issue has been resolved in the following commit: `fb53c09`.

3.3 Suggested immutable Use in AssimilatorFactory

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: AssimilatorFactory
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [2]

Description

Since version 0.6.5, Solidity introduces the feature of declaring a state as immutable. An immutable state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as immutable is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an immutable state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of immutable states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the Xave protocol, we observe there is a variable that needs not to be updated dynamically. In this case, it can be declared as immutable for gas efficiency. The related variable is the usdcAssimilator state defined in the AssimilatorFactory contract.

```
contract AssimilatorFactory is Ownable {
    event NewAssimilator(address indexed caller, bytes32 indexed id, address indexed
        assimilatorAddress);

    mapping(bytes32 => address) public assimilators;
    IOracle public immutable usdcOracle;
    IERC20 public immutable usdc;
    UsdcToUsdAssimilator public usdcAssimilator;
    ...
}
```

Listing 3.4: The States Defined in AssimilatorFactory

Recommendation Revisit the state variable definition and make good use of immutable/constant states.

Status The issue has been addressed by the following commit: fb53c09.

3.4 Caller Validation in FXPool::onJoinPool()/onExitPool()

- ID: PVE-004
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: FXPool
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In Xave, the key FXPool stablecoin pool is built on top of the BalancerV2 Vault. It is essentially composed of three main hooks that the BalancerV2 calls: onJoinPool() (upon liquidity provider deposit), onExitPool() (upon liquidity provider withdrawal), and onSwap() (upon user trade). Our analysis shows that two of them need to be revised to apply caller validation to ensure they can only be involved from the BalancerV2 Vault.

To elaborate, we show below the implementation of this onJoinPool() routine. As mentioned earlier, it is invoked when joining the pool. As a result, there is a need to ensure that it can only be called from the vault. Our analysis shows that the caller validation is not performed in the current implementation. The same is also applicable to the onExitPool() logic. Note that the onSwap() function has the proper caller validation in place.

```
    function onJoinPool(
        bytes32 poolId,
        address, // sender
        address recipient,
        uint256[] memory currentBalances, // @todo for vault transfers
        uint256,
        uint256,
        bytes calldata userData
    ) external override whenNotPaused returns (uint256[] memory amountsIn, uint256[]
        memory dueProtocolFeeAmounts) {
        (uint256 totalDepositNumeraire, address[] memory assetAddresses) = abi.decode(
            userData, (uint256, address[]));

        _enforceCap(totalDepositNumeraire);

        (uint256 lpTokens, uint256[] memory amountToDeposit) = ProportionalLiquidity.
            proportionalDeposit(
            curve,
            totalDepositNumeraire
        );

        {
            amountsIn = new uint256[](2);
            amountsIn[0] = amountToDeposit[_getAssetIndex(assetAddresses[0])];
            amountsIn[1] = amountToDeposit[_getAssetIndex(assetAddresses[1])];
```

```
        }
        curve.totalSupply = curve.totalSupply += lpTokens;

        BalancerPoolToken._mintPoolTokens(recipient, lpTokens);
        {
            dueProtocolFeeAmounts = new uint256[](2);
            dueProtocolFeeAmounts[0] = 0;
            dueProtocolFeeAmounts[1] = 0;
        }
        _mintProtocolFees();
        emit OnJoinPool(poolId, lpTokens, amountToDeposit);
    }
```

Listing 3.5: `FXPool::onJoinPool()`

Recommendation Revise the above routines to ensure the caller must be the the `BalancerV2` Vault..

Status The issue has been addressed by the following commit: `54cbdc0`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Xave` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring the fee rates as well as switching the emergency status). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setPaused() external onlyOwner {
        bool currentStatus = paused();

        if (currentStatus) {
            _unpause();
        } else {
            _pause();
        }
    }

    /// @notice Set cap for pool
    /// @param _cap cap value
```

```
    function setCap(uint256 _cap) external onlyOwner {
        (uint256 total, ) = liquidity();
        require(_cap > total, 'FXPool/cap-is-not-greater-than-total-liquidity');
        curve.cap = _cap;
    }

    /// @notice Set emergency alarm
    /// @param _emergency turn on or off
    function setEmergency(bool _emergency) external onlyOwner {
        emergency = _emergency;
        emit EmergencyAlarm(_emergency);
    }

    /// @notice Change collector address
    /// @param _collectorAddress collector's new address
    function setCollectorAddress(address _collectorAddress) external onlyOwner {
        collectorAddress = _collectorAddress;
        emit ChangeCollectorAddress(_collectorAddress);
    }

    /// @notice Change protocol percentage in fees
    /// @param _protocolPercentFee collector's new address
    function setProtocolPercentFee(uint256 _protocolPercentFee) external onlyOwner {
        protocolPercentFee = _protocolPercentFee;
        emit ProtocolFeeShareUpdated(msg.sender, protocolPercentFee);
    }
```

Listing 3.6: Example Privileged Operations in `FXPool`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 mitigated as the team confirms the use of a `multi-sig` with at least `3-of-5 majority` signatures to manage the privileged account until a governance is ready to take over.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Xave` protocol, which aims to be a one stop shop for fintechs to enable real time, cross border remittance and high yield consumer savings powered by DeFi. This audit focuses on the `AMM v2` of the `Xave` protocol, essentially composed of what is called the `FXPool` - an FX accurate stablecoin pool built on top of `Balancer`'s V2 Vault. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.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.