SMART CONTRACT AUDIT REPORT

for

Syrupal

Prepared By: Xiaomi Huang

PeckShield

September 12, 2024

Document Properties

Client	Syrupal
Title	Smart Contract Audit Report
Target	Syrupal
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 12, 2024	Xuxian Jiang	Final Release
1.0-rc	September 5, 2024	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 smart contract source code of the Syrupal 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 Syrupal

Syrupal is a cutting-edge decentralized exchange for derivatives, focusing on options and structured products. It leverages off-chain order matching, with trades executed transparently through smart contracts. Unlike other AMM-based protocols or those that price options off-chain, Syrupal is a fully on-chain options DeFi project. It implements the Black-Scholes-Merton (BSM) pricing model through smart contracts, ensuring greater transparency. Syrupal utilizes real-time price data and volatility data to ensure the accuracy of options pricing. The basic information of Syrupal is as follows:

Table 1.1: Basic Information of Syrupal

Item	Description
Target	Syrupal
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 12, 2024

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

- https://github.com/SyrupalTech/v1-core.git (a7c7e8c)

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

- https://github.com/SyrupalTech/v1-core.git (9e2433f)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

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

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

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

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

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 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 `Syrupal` implementation. 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	3	■ ■ ■
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 that 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, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Syrupal Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-002	Low	Revisited isMarketExist() Logic in PositionManager	Business Logic	Resolved
PVE-003	Low	Improved Validation of Function Arguments	Business Logic	Resolved
PVE-004	Informational	Suggested Adherence of Checks-Effects-Interactions	Time and State	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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Delegate`
- Category: Coding Practices [6]
- 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).

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

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

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

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()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.2: `SafeERC20::safeApprove()`

In current implementation, if we examine the `InsuranceFund::constructor()` routine, it is used to initially approve the spending allowance to the `USDC` contract. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (line 54).

```
48    constructor(IPositionManager _manager, IUSDX _usdx, address _operator) {
49        manager = _manager;
50        usdx = _usdx;
51        operator = _operator;
52
53        stablecoin = usdx.stablecoin();
54        stablecoin.approve(address(_usdx), type(uint256).max);
55    }
```

Listing 3.3: `InsuranceFund::constructor()`

Note the resetApproval() routine in the same contract can be similarly improved.

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

Status The issue has been addressed in the following commit: 9e2433f.

3.2 Revisited isMarketExist() Logic in PositionManager

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PositionManager
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Syrupal protocol has a core PositionManager contract that manages active markets as well as each user SubAccount as a unique ERC721 token ID. Naturally, it provides a number of helper routines to access active markets and user accounts. While examining the logic behind a specific isMarketExist() routine, we notice it can be improved by thoroughly validating all possible input cases.

To elaborate, we show below the implementation of the related isMarketExist() routine. As the name indicates, this routine is designed to check whether a given market ID exists. It comes to our attention that the implementation misses the corner case about market 0. And market 0 should not be consider as present, which requires to revise the logic to be if (marketId > lastMarketId || marketId == 0)revert Errors.NotExistMarket().

```
    /// @notice Check whether a given market ID exists
    /// @param marketId The market ID to check
    function isMarketExist(uint256 marketId) public view {
        if (marketId > lastMarketId) revert Errors.NotExistMarket();
    }
```

Listing 3.4: PositionManager::isMarketExist()

Recommendation Revise the above logic of isMarketExist() to properly check whether the given market ID exists.

Status The issue has been addressed in the following commit: 9e2433f.

3.3 Improved Validation on Function Arguments

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PositionManager`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Syrupal` protocol is no exception. Specifically, if we examine the `PositionManager` contract, it has defined a number of protocol-wide risk parameters, such as `IMLowerRatio` and `IMUpperRatio` . In the following, we show the corresponding routines that allow for their changes.

```
    function setMarginParams(uint256 marketId, MarginParams calldata params) external
        onlyOperator {
        isMarketExist(marketId);

        if (params.IMUpperRatio > UNIT  params.IMLowerRatio > UNIT  params.MMRatio >
            UNIT) {
            revert Errors.InvalidMarginParams();
        }

        marginParams[marketId] = params;

        emit MarginParamsSet(marketId, params.IMUpperRatio, params.IMLowerRatio, params.
            MMRatio);
    }
```

Listing 3.5: PositionManager::setMarginParams()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, the above routine can be improved by further enforcing the following requirement: `params.IMLowerRatio < params.IMUpperRatio`.

Moreover, in the `USDX` contract, the `deposit()` function can be improved by validating the given `recipientAccount` is legitimate and have its owner. The `withdraw()` function can be improved to ensure the receive is not `address(0)`, i.e., `require(receiver != address(0))`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been addressed in the following commit: `9e2433f`.

3.4 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `USDX`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the `Uniswap/Lendf.Me` hack [12].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `USDX` as an example, the `_withdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 199) starts before effecting the update on internal state (lines 201-209), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `_withdraw()` function. Note that there is no harm that may be caused to current protocol. However, it is still suggested to follow the known `checks-effects-interactions` best practice. Note the `deposit()` routine can be similarly improved.

```
184     function _withdraw(uint256 accountId, uint256 amount, address recipient) internal {
185         uint256 exchangeRate = _getExchangeRate();
186 
187         uint256 stableAmount = amount.multiplyDecimal(exchangeRate).from18Decimals(
            stableDecimal);
188 
189         stablecoin.safeTransfer(recipient, stableAmount);
190 
191         _balanceAdjustment(
192             BalanceAdjustment({
193                 accountId: accountId,
194                 asset: IUSDX(address(this)),
195                 subId: 0,
196                 amount: -(amount.toInt256())
197             }),
```

```
            ""
        );

        emit Withdraw(accountId, recipient, amount, stableAmount);
    }
```

Listing 3.6: USDX::_withdraw()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice.

Status The issue has been addressed in the following commit: `9e2433f`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Syrupal` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters and assigning various roles). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
    function setOperator(address _newOperator) external onlyOwner {...}
    ...
    function setLiquidator(address _liquidator, bool _trusted) external onlyOperator
        {...}
    ...
    function setExecutor(address _executor, bool _trusted) external onlyOperator {...}
    ...
    function setDiscount(uint64 _discount) external onlyOperator {...}
    ...
    function setMarketOracles(
        uint256 marketId,
        ISpotOracle spotOracle,
        IForwardOracle forwardOracle,
        IVolatilityOracle volOracle
    ) external onlyOperator {...}
```

```
    ...
    function setStableOracle(ISpotOracle _stableOracle) external onlyOperator {...}
    ...
    function setMarginParams(uint256 marketId, MarginParams calldata params) external
        onlyOperator {...}
    ...
    function setDepegParams(DepegParams calldata params) external onlyOperator {...}
    ...
    function setFeeRecipient(uint256 _newRecipient) external onlyOperator {...}
    ...
    function setFeeExemption(address caller, bool exempted) external onlyOperator {...}
```

Listing 3.7: Example Privileged Operations in `Factory`

We emphasize that the privilege assignment is necessary and consistent with the protocol design. However, it is worrisome if the `owner` 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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the `owner`.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Syrupal`, which is a cutting-edge decentralized exchange for derivatives, focusing on options and structured products. It leverages off-chain order matching, with trades executed transparently through smart contracts. Unlike other `AMM`-based protocols or those that price options off-chain, `Syrupal` is a fully on-chain options `DeFi` project. It implements the `Black-Scholes-Merton (BSM)` pricing model through smart contracts, ensuring greater transparency. `Syrupal` utilizes real-time price data and volatility data to ensure the accuracy of options pricing. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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.

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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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 CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.