SMART CONTRACT AUDIT REPORT

for

SYNFUTURES

Prepared By: Shuxiao Wang

Hangzhou, China
December 18, 2020

Document Properties

Client	SynFutures
Title	Smart Contract Audit Report
Target	SynFutures
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xudong Shao, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	December 18, 2020	Xuxian Jiang	Final Release
1.0-rc1	December 15, 2020	Xuxian Jiang	Release Candidate #1
0.3	December 7, 2020	Xuxian Jiang	Add More Findings #2
0.2	December 5, 2020	Xuxian Jiang	Add More Findings #1
0.1	December 3, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the **SynFutures** 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 SynFutures

SynFutures is an open and decentralized derivatives platform that allows a variety of assets, including Ethereum native, cross-chain and off-chain real world assets to be synthesized and freely traded. In the first version of the contract, SynFutures will launch a digital asset futures market to introduce (1) futures contract of arbitrary assets and expiration dates to be created by liquidity providers, (2) Synthetic Automated Market Maker (sAMM), for market participants to provide one single digital asset of a trading pair only and the smart contract to synthesize the other, and (3) Automated Liquidator (ALQ), which reduces the entry barrier of liquidators and helps automate the liquidation process. `SynFutures` presents an interesting addition of innovation to current DeFi ecosystem.

The basic information of SynFutures is as follows:

Table 1.1: Basic Information of SynFutures

Item	Description
Target	SynFuturesV1
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 18, 2020

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

- https://github.com/SynFutures/synfutures-contract-v1 (798e63f)

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

- https://github.com/SynFutures/synfutures-contract-v1 (f693fdc)

1.2 About PeckShield

PeckShield Inc. [14] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 the OWASP Risk Rating Methodology [13]:

- 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.

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

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 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) [12], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

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

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 SynFutures 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 logics, 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	2	■■
Low	5	■■■■■
Informational	2	■■
Total	9	

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 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, 2 informational recommendations.

Table 2.1: Key Audit Findings of SynFutures Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Miscalculation of _alignExpiry()	Numeric Errors	Fixed
PVE-002	Low	Quote Adjustment And Friday Alignment in Reader	Business Logic	Fixed
PVE-003	Medium	Remove Unnecessary Admin Rights	Security Features	Mitigated
PVE-004	Low	Improved Calculation of leftDays in depositAndInitPool()	Numeric Errors	Fixed
PVE-005	Low	Unchangeable Feeders After ChainlinkOracle Initialization	Business Logic	Fixed
PVE-006	Informational	Inconsistency Between Documentation and Implementation	Coding Practices	Fixed
PVE-007	Low	Improved transferFrom() in ShareToken	Business Logic	Fixed
PVE-008	Informational	Unused Code Removal	Coding Practices	Fixed
PVE-009	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Miscalculation of _alignExpiry()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Factory
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [3]

Description

SynFutures aims to build a futures market with arbitrary asset and expiration date that can be determined by liquidity providers. The proposed synthetic automated market maker (sAMM) model allows for similar trading experience for futures margin trading. Meanwhile, for the same pair of base and quote assets, different expiration date creates different futures contracts that however lead to the provided liquidity scattered. To mitigate this issue, SynFutures pools the scattered liquidity by aligning the expiry times of futures contracts to the corresponding 8:00am (UTC) on Friday of the same week of the original expiry times.

To this end, the `Factory` contract provides a helper routine namely `_alignExpiry()`. For illustration, we show below its full implementation. If the supported asset is configured for expiry alignment, it adjusts the expiry time to be 08:00:00 Friday (UTC time) of the week the original expiry lies in.

```
    function _alignExpiry(uint _expiry, Types.MarginParam memory param) internal view
        returns (uint expiry) {
        // solium-disable-next-line security/no-block-members
        require(_expiry > block.timestamp + 1 hours, "_alignExpiry: bad _expiry");
        if (param.alignToFriday) {// align to 08:00:00 Friday (UTC time) of the week
            _expiry lies in
            // block.timestamp is seconds since unix epoch 1970/01/01/00:00:00, which is
                00:00:00 Thursday
            expiry = (_expiry - 4 days) / 1 weeks * 1 weeks + 4 days + 8 hours;
        } else { // align to UTC 08:00:00 of the day _expiry lies in
            expiry = (_expiry) / 1 days * 1 days + 8 hours;
        }
```

```
133         // solium-disable-next-line security/no-block-members
134         require(expiry > block.timestamp + 1 hours, "_alignExpiry: bad _expiry");
135     }
```

Listing 3.1: Factory :: _alignExpiry()

However, our analysis shows that the aligned expiry is miscalculated. The original calculation of `expiry = (_expiry - 4 days)/ 1 weeks * 1 weeks + 4 days + 8 hours` (line 129) forgets to adjust the initial offset as the very first `Unix epoch` starts on Thursday. Therefore, the proper adjustment should be the following: `expiry = (_expiry + 3 days)/ 1 weeks * 1 weeks + 4 days + 8 hours - 3 days`.

Recommendation Revise the `_alignExpiry()` logic to return the proper expiry date after alignment. An example revision is shown below.

```
124     function _alignExpiry(uint _expiry, Types.MarginParam memory param) internal view
            returns (uint expiry) {
125         // solium-disable-next-line security/no-block-members
126         require(_expiry > block.timestamp + 1 hours, "_alignExpiry: bad _expiry");
127         if (param.alignToFriday) {// align to 08:00:00 Friday (UTC time) of the week
                _expiry lies in
128             // block.timestamp is seconds since unix epoch 1970/01/01/00:00:00, which is
                    00:00:00 Thursday
129             expiry = (_expiry + 3 days) / 1 weeks * 1 weeks + 4 days + 8 hours - 3 days;
130         } else { // align to UTC 08:00:00 of the day _expiry lies in
131             expiry = (_expiry) / 1 days * 1 days + 8 hours;
132         }
133         // solium-disable-next-line security/no-block-members
134         require(expiry > block.timestamp + 1 hours, "_alignExpiry: bad _expiry");
135     }
```

Listing 3.2: Factory :: _alignExpiry()

Status This issue has been fixed in this commit: `cb7156f`.

3.2 Quote Adjustment And Friday Alignment in Reader

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `Reader`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

SynFutures is equipped with a `Reader` contract to expose its run-time states. In particular, it allows external entities to query the current contract addresses of both `AMM` and `Futures` for a given pair of base and quote assets as well as the associated expiry. It also enables the query of the details of a running `AMM`, `Futures`, or a given account.

While examining various query handlers, we notice a specific one, i.e., `getChainlinkContractAddresses()`. This specific handler takes `factoryAddr`, `base`, `quote`, and `expiry` as arguments and returns the corresponding `AMM` and `Futures`. However, the current routine uses a hardcoded `USDC` as the quote asset (line 35), instead of the given `quote` argument. Apparently, if the pair does not use `USDC` as the quote asset, this routine returns the wrong information of `AMM` and `Futures`.

```
29      function getChainlinkContractAddresses(
30          address factoryAddr, string memory base, address quote, uint expiry
31      ) public view returns (address ammProxy, address futuresProxy) {
32          Factory factory = Factory(factoryAddr);

34          address oracle = factory.oracleController().getChainlinkOracle(base, quote);
35          bytes32 index = keccak256(abi.encodePacked(oracle, factory.USDC, expiry));
36          futuresProxy = factory.pairsForFutures(index);
37          ammProxy = factory.pairsForAmm(index);
38      }
```

Listing 3.3: Reader::getChainlinkContractAddresses()

Recommendation Modify the `getChainlinkContractAddresses()` logic to use the given `quote` argument. An example revision is shown below.

```
    function getChainlinkContractAddresses(
        address factoryAddr, string memory base, address quote, uint expiry
    ) public view returns (address ammProxy, address futuresProxy) {
        Factory factory = Factory(factoryAddr);

        address oracle = factory.oracleController().getChainlinkOracle(base, quote);
        bytes32 index = keccak256(abi.encodePacked(oracle, quote, expiry));
        futuresProxy = factory.pairsForFutures(index);
        ammProxy = factory.pairsForAmm(index);
```

```
    }
```

Listing 3.4: Reader::getChainlinkContractAddresses()

Status This issue has been fixed in this commit: `cb7156f`.

3.3 Remove Unnecessary Admin Rights

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

In SynFutures, there is a privileged account, i.e., `owner`, that plays a critical role in not only governing and regulating the system-wide operations (e.g., margin asset addition and parameter setting), but also managing each trader's account (and the balance directly determines the withdrawable assets for each trader).

If we take a close look at the `Storage` contract, this specific contract takes a number of routines, e.g., `increaseAccountBalance()`, `decreaseAccountBalance()`, and `setSocialLossPerContract()`. The first two routines can be used to adjust the trader balance while the last one is used to directly set the social loss for a specified side. These are privileged routines governed by the `onlyOwner` modifier.

```
    function setSocialLossPerContract(Types.Side side, int newVal) public onlyOwner
        onlyEmergency {
        require(side == Types.Side.LONG  side == Types.Side.SHORT, "unknown side");
        socialLossPerContracts[uint(side)] = newVal;
        emit UpdateSocialLoss(side, newVal);
    }

    // Set cash balance of account in emergency by admin
    function increaseAccountBalance(address trader, uint amount) public onlyOwner
        onlyEmergency {
        _updateAccountBalance(trader, amount.toInt256());
    }

    // Set cash balance of account in emergency by admin
    function decreaseAccountBalance(address trader, uint amount) public onlyOwner
        onlyEmergency {
        _updateAccountBalance(trader, amount.toInt256().neg());
    }
```

Listing 3.5: Storage.sol

Also, the `owner` can set the states of `futures` contracts to be `Emergency`, which immediately terminates the contracts with a given `settle` price.

As a mitigation, instead of having a single EOA account as the `Owner`, an alternative is to make use of a multi-sig wallet. To further eliminate the administration key concern, it may be required to transfer the role to a community-governed DAO. In the meantime, a timelock-based mechanism might also be applicable for mitigation.

Recommendation Promptly transfer the `Owner` privilege to the intended `DAO`-like governance contract. And 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 by removing the above-mentioned functions. Also, the team plans to transfer the privilege to the intended `DAO`-like governance after the mainnet deployment becomes stable and mature.

3.4 Improved Calculation of leftDays in depositAndInitPool()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Amm`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [3]

Description

In Section 3.3, we have examined the `depositAndInitPool()` routine and elaborate an issue with manipulated `index price`. In this section, we focus on the same routine and analyze the enforcement of the given `initPrice`.

The `initPrice` is provided by the first liquidity provider in setting up the initial `mark price` for the intended futures contract. To ensure this `initPrice` falls in a proper price range, SynFutures has a system-wide risk parameter, i.e., `maxInitialDailyBasis`. Based on the number of remaining days to the expiry of the related futures contract, the protocol can compute the maximum allowed deviation of `initPrice`.

To elaborate, we show below the code snippet of `depositAndInitPool()`. As mentioned earlier, this routine is used by the very first liquidity provider to add liquidity into the pool.

```
    // deposit margin and initialize the pool, margin token amount and leverage in 10^18
    function depositAndInitPool(uint wadAmount, uint initPrice, uint leverage, uint
        deadline) public payable {
        require(_getBlockTimestamp() <= deadline, "deadline exceeded");
```

```
205         require(status == Types.Status.NORMAL && _getBlockTimestamp() <= expiry - 1 hours, "bad status");
206         require(futuresProxy.getAccount(address(this)).position == 0, "pool not empty");
207         require(wadAmount > 0 && initPrice > 0, "bad wadAmount/initial price");

209         uint ONE = LibMathUnsigned.WAD();
210         Types.Param memory param = config.parameter();
211         // validate leverage range
212         require(leverage >= ONE && leverage.wmul(_c2w(param.initialMarginRatio)) <= ONE, "bad leverage");
213         {
214             uint blockTime = _getBlockTimestamp();
215             // validate maxInitialDailyBasis
216             uint price = indexPrice(); // initial price < index price +/- (days * max daily basis)
217             uint basis = initPrice > price ? (initPrice - price) : (price - initPrice);
218             uint leftDays = (expiry - blockTime) / 1 days + 1;
219             uint maxBasis = price.wmul(_c2w(param.maxInitialDailyBasis)).mul(leftDays);
220             require (basis <= maxBasis, "bad initPrice");

222             // init markPriceState
223             require(markPriceState.lastMarkTime == 0, "already initialized");
224             markPriceState.lastMarkTime = uint32(blockTime);
225             markPriceState.lastIndexPrice = uint112(price);
226             markPriceState.lastEmaBasis = int112(initPrice.toInt256().sub(price.toInt256()));
227         }
228         // use truncated (probably) wadAmount returned by depositFor for later calculation
229         wadAmount = futuresProxy.depositFor {value : msg.value} (msg.sender, wadAmount); // deposit for trader

231         // wadAmount = price * size * 2 + price * size / leverage
232         // size = wadAmount / (2 * price + price / leverage)
233         // ONE <= leverage
234         uint denominator = initPrice.mul(2).add(initPrice.wdiv(leverage));
235         uint size = wadAmount.wdiv(denominator);
236         // to prevent precision error caused by rounding
237         if (size.wmul(denominator) > wadAmount) size = size.sub(1);

239         // trade with AMM and transfer margin
240         futuresProxy.tradeWithMarginFor(msg.sender, Types.Side.SHORT, initPrice, size, true);
241         _mint(msg.sender, size);
242         // the calculation above already makes sure that both trader and amm's accounts are safe after trade
243     }
```

Listing 3.6: Amm::depositAndInitPool()

The related enforcement of `initPrice` occurs at line 220: `require (basis <= maxBasis, "bad initPrice")`. The `basis` in essence is the computed difference of `initPrice` when compared with the current `index price`, while `maxBasis` is calculated as `maxBasis = price.wmul(_c2w(param.maxInitialDailyBasis)).mul(leftDays)` (line 219). The `maxInitialDailyBasis` is the system-wide risk parameter that regulates the maximum daily price deviation permitted by the protocol. Note the `leftDays` is computed as `leftDays = (expiry - blockTime)/ 1 days + 1` (line 218). There is an off-by-one bug in the `leftDays` calculation. The correct one is `leftDays = (expiry - blockTime - 1)/ 1 days + 1`.

Recommendation Change current execution logic of `depositAndInitPool()` to properly calculate `leftDays` in order to validate the given `initPrice`.

Status This issue has been fixed in this commit: `47e4398`.

3.5 Unchangeable Feeders After ChainlinkOracle Initialization

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OracleController`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

SynFutures supports both `Uniswap` and `Chainlink` as its oracles. For a given pair of base and quote assets, the `OracleController` contract provides two different methods to generate the requested oracle: one method is `newChainlinkOracle()` with `Chainlink` price feeds and the another is `newUniswapOracle()` from the built-in AMM price curve.

In the following, we examine the `newChainlinkOracle()` routine and shows its code snippet below. Its execution logic is as follows: it firstly validates that the intended oracle has not been created (line 95) and the protocol allows its creation (line 96), then validates the presence of `Chainlink feeder`, next instantiates a new `ChainlinkOracle` with the validated `Chainlink feeder` (line 106), and finally emits the related event for the oracle creation (line 109).

```
94    function newChainlinkOracle(string memory base, address quote) public returns (
          address) {
95        require(chainlinkOracles[base][quote] == address(0), "newChainlinkOracle: oracle
              already exists");
96        Types.MarginParam memory param = config.marginsParam(quote);
97        require(param.allowed, "newChainlinkOracle: unsupported quote");

99        address feeder = chainlinkFeeders[base][quote];
100       require(feeder != address(0), "newChainlinkOracle: no price feeder for base/
              quote");

102       uint decimal = IChainlinkAggregator(feeder).decimals();
```

```
        require(decimal <= 18, "newChainlinkOracle: chainlink aggregator's decimal exceeds 18");
        uint scaler = 10**(18 - decimal);

        address oracle = address(new ChainlinkOracle(feeder, scaler));
        chainlinkOracles[base][quote] = oracle;

        emit NewChainlinkOracle(base, quote, oracle);
        return oracle;
    }

    function updateChainlinkFeeder(string memory base, address quote, address feeder)
        public onlyOwner {
        chainlinkFeeders[base][quote] = feeder;
        emit UpdateChainlinkFeeder(base, quote, feeder);
    }
```

Listing 3.7: OracleController :: newChainlinkOracle()

From the above execution logic, we notice that the `Chainlink` oracle relies on the presence of an existing `Chainlink feeder`. Meanwhile, we also notice the presence of a `updateChainlinkFeeder()` routine to update the `Chainlink feeder`. However, the updated `Chainlink feeder` has no effect on already created `Chainlink` oracles.

Recommendation Revise the `updateChainlinkFeeder()` logic to ensure the absence of the given `Chainlink feeder`. An example revision is shown below.

```
    function updateChainlinkFeeder(string memory base, address quote, address feeder)
        public onlyOwner {
        require(chainlinkFeeders[base][quote] == address(0), "!feeder already exists" )
        chainlinkFeeders[base][quote] = feeder;
        emit UpdateChainlinkFeeder(base, quote, feeder);
    }
```

Listing 3.8: OracleController :: updateChainlinkFeeder()

Status This issue has been fixed in this commit: `47e4398`.

3.6 Inconsistency Between Documentation and Implementation

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

Specifically, the `Factory` contract maintains the states of `pairsIndex`, `pairsForAmm` and `pairsForFutures`. The `pairsIndex` array keeps all pair indexes, `pairsForAmm` records the mapping from a pair index to the corresponding `sAMM`; and `pairsForFutures` represents the mapping from a pair index to the corresponding `futures` contract.

In the following, we show their definitions (see the code snippet below). It comes to our attention that the pair index is indicated as `keccak[base|quote|oracle|expiry]` (lines $29-31$).

```
15 contract Factory is Ownable {
16     using Types for Types.MarginParam;
17
18     bytes4 private constant _DECIMAL_SELECTOR = bytes4(keccak256(bytes("decimals()")));
19     bytes4 private constant _SYMBOL_SELECTOR = bytes4(keccak256(bytes("symbol()")));
20
21     address public immutable USDC;
22
23     IGlobalConfig public config;
24     address public futuresLogic;
25     address public uniswapAmmLogic;
26     address public chainlinkAmmLogic;
27     IOracleController public oracleController;
28
29     mapping(bytes32 => address) public pairsForAmm; // keccak[basequoteoracleexpiry] ->
          AmmProxymapping(bytes32 => address) public pairsForFutures; //
          keccak[basequoteoracleexpiry] --> FuturesProxy
30
31     bytes32[] public pairsIndex; // book all pairs index: keccak[basequoteoracleexpiry]...
```

Listing 3.9: Factory.sol

However, if we pay attention to the actual `_createAndBookProxies()` routine that computes the index (line 108), the proper index is `keccak256(abi.encodePacked(oracle, quote, alignedExpiry))`, which is inconsistent with the above definition.

```
105     function _createAndBookProxies(
106         string memory name, address quote, address oracle, uint alignedExpiry
107     ) internal returns (FuturesProxy, AmmProxy) {
108         bytes32 index = keccak256(abi.encodePacked(oracle, quote, alignedExpiry));
109         require(pairsForAmm[index] == address(0) && pairsForFutures[index] == address(0)
                ,
110                 "_createAndBookProxies: pair already exists");
111
112         uint8 decimal = _getDecimal(quote);
113         FuturesProxy futuresProxy = new FuturesProxy(futuresLogic, address(config),
                quote, decimal);
114         address ammLogic = (IOracle(oracle).class() == Types.Oracle.UNISWAP) ?
                uniswapAmmLogic : chainlinkAmmLogic;
```

```
        AmmProxy ammProxy = new AmmProxy(ammLogic, address(config), address(futuresProxy
            ), oracle, alignedExpiry, name);

        pairsForFutures[index] = address(futuresProxy);
        pairsForAmm[index] = address(ammProxy);
        pairsIndex.push(index);

        return (futuresProxy, ammProxy);
    }
```

Listing 3.10: Factory::_createAndBookProxies()

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been fixed in this commit: `47e4398`.

3.7 Improved transferFrom() in ShareToken

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ShareToken`
- Category: Business Logic [10]
- CWE subcategory: CWE-754 [6]

Description

In SynFutures, the `AMM` or more precisely `ShareToken` is an ERC20-compliant pool token that represent the ownership of liquidity providers in the shared pool. Accordingly, there is a need for the pool token contract implementation, i.e., `ShareToken`, to follow the ERC20 specification. In the following, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic.

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited SynFutures. In the following two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification.

Meanwhile, we notice in the `transferFrom()` routine, there is a common practice that is missing but widely used in other ERC20 contracts. Specifically, when `msg.sender = _from`, the current `transferFrom()` implementation disallows the token transfer if `msg.sender` has not explicitly allows spending from herself yet. A common practice will whitelist this special case and allow `transferFrom()` if `msg.sender = _from` even there is no allowance specified.

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

```
    function transferFrom(address from, address to, uint value) external returns (bool) {
        if (allowance[from][msg.sender] != uint(-1)) {
            allowance[from][msg.sender] = allowance[from][msg.sender].sub(value);
        }
        _transfer(from, to, value);
        return true;
    }
```

Listing 3.11: ShareToken.sol

Recommendation Improve the `transferFrom()` logic by considering the special case when `msg.sender = _from`. In the meantime, consider the support of `permit()` (in EIP-2612) for better integration and usability.

Status This issue has been fixed in this commit: `47e4398`.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approve() event	Is emitted on any successful call to approve()	✓

3.8 Unused Code Removal

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LibMathSigned, FuturesProxy
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [5]

Description

SynFutures makes good use of a number of reference contracts, such as ERC20, Math, SafeMath, SignedSafeMath, and SafeCast, to facilitate its code implementation and organization. For example, the Account smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the LibMathSigned contract (see the code snippet below), there are a number of constants that are defined, but not used. Specifically, the following constants are not used in the current code base: FIXED_DIGITS, FIXED_1, FIXED_E, LONGER_DIGITS, LONGER_FIXED_LOG_E_1_5, LONGER_FIXED_1, and LONGER_FIXED_LOG_E_10.

```
library LibMathSigned {
    using SignedSafeMath for int;

    int private constant _WAD = 10 ** 18;
    int private constant _INT256_MIN = -2 ** 255;

    uint8 private constant FIXED_DIGITS = 18;
    int private constant FIXED_1 = 10 ** 18;
    int private constant FIXED_E = 2718281828459045235;
    uint8 private constant LONGER_DIGITS = 36;
    int private constant LONGER_FIXED_LOG_E_1_5 = 405465108108164381978013115464349137;
    int private constant LONGER_FIXED_1 = 10 ** 36;
    int private constant LONGER_FIXED_LOG_E_10 = 2302585092994045684017991454684364208;
    ...
}
```

Listing 3.12: LibMathSigned.sol

In the meantime, the setAmm() routine in FuturesProxy is also unnecessary as the fallback() routine is already in place for the same result.

Recommendation Remove the unused constants and the redundant setAmm() routine in FuturesProxy.

Status This issue has been fixed in this commit: 47e4398.

3.9 Improved Sanity Checks For System Parameters

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GlobalConfig`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The SynFutures protocol is no exception. Specifically, if we examine the `GlobalConfig` contract, it has defined the system-wide risk parameters with the type: `Types.Param`. This type consolidates all system-wide risk parameters, as shown below.

```
    struct Param { // only takes 1 slot
        uint32 emaTimeConstant; // in seconds, max 86400
        // all ratios are less than 1, and is scaled by 10000, e.g. 0.01 --> 100, 0.2 --> 2000
        uint16 poolFeeRatio;
        uint16 poolDevFeeRatio;
        // maximum price dislocation in a single block for either direction from the mid price at the start of the current block
        // this serves as a speed limit of price move for the AMM and protects the system from attacks involving distorting the market with in the same block
        // a trade would be reverted if it result in a price move of this block breaching this limit
        uint16 maxPriceSlippageRatio;
        // maximum deviation of initial price to spot index per day to limit the initial price for AMM in a reasonable range
        uint16 maxInitialDailyBasis; // initPrice - indexPrice < indexPrice * days * maxInitialDailyBasis
        // maximum open interest ratio of the entire market for a single user(address) to prevent concentration of risk in a single account
        // when a user's account has high open interest ratio than this limit, the user can only execute trades to reduce position but not increase position
        // this limit does not apply to the action of LP adding liquidity to the AMM
        // but if a LP's position breaches the limit after adding liquidity to the AMM, they cannot increase their position further
        uint16 maxUserTradeOpenInterestRatio;
        // minimum open interest ratio of the entire market for the AMM to prevent a drain of liquidity
        // the AMM needs to maintain certain level of inventory to prevent large slippages as every user can only trade with the AMM
        // this limit applies to both users buying from the AMM and LPs removing liquidity
        uint16 minAmmOpenInterestRatio;
        // maximum spot index change from oracle that can be accepted since the last update
```

```
        // as mark price are updated at most once per block this serves as a speed limit of the mark price
        // and protects the system from attacks involving distorting the underlying oracle in a short period of time
        uint16 maxSpotIndexChangePerSecondRatio;
        uint16 initialMarginRatio;
        uint16 maintenanceMarginRatio;
        // used when a liquidated account is already bankrupt and thus no remaining maintenance margin can be used to reward the liquidator
        // the reward would be withdraw from insurance fund in this case to keep liquidators motivated
        uint16 bankruptcyLiquidatorRewardRatio;
        uint16 insurancePremiumRatio;
    }
```

Listing 3.13: LibTypes::Param

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, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `maxPriceSlippageRatio` may allow unreasonably large slippage for the futures trades.

To elaborate, we show below its code snippet of `setParameter()`. This routine updates various parameters defined in `Types.Param`. However, they can be improved to validate that the given arguments. For example, `_minAmmOpenInterestRatio` can be no more than 5% and `_maxUserTradeOpenInterestRatio` can be restrictive as well in the range of $[5\% - 10\%]$.

```
    function setParameter(bytes32 key, uint value) public onlyOwner {
        if (key == "emaTimeConstant") {
            require(value <= 86400, "emaTimeConstant cannot exceed 86400");
            parameter.emaTimeConstant = value.toUint32();
        } else {
            require(value < 10000, "ratio must < 1");
            uint16 ratio = uint16(value); // no need to user .toUint16()

            if (key == "poolFeeRatio") {
                parameter.poolFeeRatio = ratio;
            } else if (key == "poolDevFeeRatio") {
                parameter.poolDevFeeRatio = ratio;
            } else if (key == "maxPriceSlippageRatio") {
                parameter.maxPriceSlippageRatio = ratio;
            } else if (key == "maxInitialDailyBasis") {
                parameter.maxInitialDailyBasis = ratio;
            } else if (key == "maxUserTradeOpenInterestRatio") {
                parameter.maxUserTradeOpenInterestRatio = ratio;
            } else if (key == "minAmmOpenInterestRatio") {
                parameter.minAmmOpenInterestRatio = ratio;
            } else if (key == "maxSpotIndexChangePerSecondRatio") {
```

```
                parameter.maxSpotIndexChangePerSecondRatio = ratio;
            } else if (key == "initialMarginRatio") {
                require(parameter.maintenanceMarginRatio < ratio, "require mm < im");
                parameter.initialMarginRatio = ratio;
            } else if (key == "maintenanceMarginRatio") {
                require(parameter.insurancePremiumRatio < ratio, "require pfr < mm");
                require(ratio < parameter.initialMarginRatio, "require mm < im");
                parameter.maintenanceMarginRatio = ratio;
            } else if (key == "insurancePremiumRatio") {
                require(ratio < parameter.maintenanceMarginRatio, "require ip < mm");
                require(parameter.bankruptcyLiquidatorRewardRatio < ratio, "require blr < ip");
                parameter.insurancePremiumRatio = ratio;
            } else if (key == "bankruptcyLiquidatorRewardRatio") {
                require(ratio < parameter.insurancePremiumRatio, "require blr < ip");
                parameter.bankruptcyLiquidatorRewardRatio = ratio;
            } else {
                revert("key not exists");
            }
        }
        emit UpdateParameter(key, value);
    }
```

Listing 3.14: GlobalConfig::setParameter()

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status This issue has been fixed in this commit: dd0f132.

4 Conclusion

In this audit, we have analyzed the design and implementation of SynFutures, an open and decentralized derivatives platform that allows a variety of assets, including Ethereum native, cross-chain and off-chain real world assets to be synthesized and freely traded. The system presents a clean and consistent design that makes it a distinctive and valuable addition of innovation to current DeFi ecosystem. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[5] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[6] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

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

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

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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