SMART CONTRACT AUDIT REPORT

for

Sense Protocol

Prepared By: Yiqun Chen

PeckShield

November 7, 2021

Document Properties

Client	Sense Protocol
Title	Smart Contract Audit Report
Target	Sense Protocol
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Patrick Liu, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	November 7, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
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 `Sense` 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 Sense

`Sense` is a decentralized fixed-income protocol on `Ethereum`, allowing users to manage risk through fixed rates and future yield trading on existing yield bearing-assets. It allows users to decompose a yield-bearing asset into its principal and yield components and package them behind two maturing assets, a `Zero` and a `Claim`. With `Zeros` and `Claims`, users can safely earn/borrow at a fixed rate and trade against future yields without the risk of liquidation and capital lock-ups.

The basic information of the `Sense` protocol is as follows:

Table 1.1: Basic Information of The `Sense` Protocol

Item	Description
Name	Sense Protocol
Website	https://sense.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 7, 2021

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

- https://github.com/sense-finance/sense-v1.git (e0be5ae)

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

- https://github.com/sense-finance/sense-v1.git (b65a1c)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

Impact \ 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 [9]:

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

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

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

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

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 `Sense` 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	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, and 3 low-severity vulnerabilities, and 1 informational recommendations.

Table 2.1: Key Sense Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested SafeMath For Overflow Prevention	Coding Practices	Fixed
PVE-002	Low	Improved Precision By Multiplication And Division Reordering	Coding Practices	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Suggested immutable in EmergencyStop/GClaimManager	Coding Practices	Fixed
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	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 Suggested SafeMath For Overflow Prevention

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: GClaimManager
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

SafeMath is a Solidity math library especially designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. Our analysis shows that the current implementation can be improved with the use of SafeMath.

Specifically, we show below the excess() function from the GClaimManager contract. This function is used to calculate the amount of excess that has accrued since the first Claim from a Series was deposited. However, it is possible that the internal computation of currScale - initScale may result in an underflow.

```
    /// @notice Calculates the amount of excess that has accrued since the first Claim
        from a Series was deposited
    function excess(
        address feed,
        uint256 maturity,
        uint256 balance
    ) public returns (uint256 amount) {
        (, address claim, , , , , , , ) = Divider(divider).series(feed, maturity);
        uint256 initScale = inits[claim];
        uint256 currScale = Feed(feed).scale();
        if (currScale - initScale > 0) {
            amount = (balance * currScale) / (currScale - initScale) / 10**18;
        }
    }
```

Listing 3.1: GClaimManager::excess()

Moreover, the `OracleLibrary::consult()` function can be improved to validate the calculation of an internal variable named `timeWeightedAverageTick` from the following statement `timeWeightedAverageTick = int24(tickCumulativesDelta / int56(int32(period)))` does not result in any type casting error.

Recommendation Revise the above logic to properly validate the given input to avoid unnecessary overflow computation.

Status The issue has been fixed by this commit: `635aaaa`.

3.2 Improved Precision By Multiplication And Division Reordering

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GClaimManager`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

As mentioned in Section 3.1, `SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, if we examine the `GClaimManager::exit()` routine, this routine has internal computation of `rights` that involves a number of operators, i.e., `balance.fdiv(gclaims[claim].totalSupply(), Token(claim).BASE_UNIT()).fmul(total, Token(claim).BASE_UNIT())` (line 92). This computation can be simplified as `balance.fdiv(gclaims[claim].totalSupply(), total)` by canceling out common factors in both denominators and numerators.

```
75    function exit(
76        address feed,
77        uint256 maturity,
78        uint256 balance
79    ) external {
80        (, address claim, , , , , , ) = divider.series(feed, maturity);

82        require(claim != address(0), Errors.SeriesDoesntExists);

84        // Collect excess for all Claims from this Series in this contract holds
```

```
        uint256 collected = Claim(claim).collect();
        // Track the total Target collected manually so that that we don't get
        // mixed up when multiple Series have the same Target
        uint256 total = totals[claim] + collected;

        // Determine the percent of the excess this caller has a right to
        uint256 rights = (
            balance.fdiv(gclaims[claim].totalSupply(), Token(claim).BASE_UNIT()).fmul(
                total, Token(claim).BASE_UNIT())
        );
        total -= rights;
        totals[claim] = total;
        ...
    }
```

Listing 3.2: GClaimManager::exit()

Similar optimizations can also be applicable to the calculation of `tBalZeroIdeal` and `tBalZeroActual` inside the `Divider::_redeemClaim()` function. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed by this commit: `635aaaa`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Sense` protocol, there is a privileged `isTrusted` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and feed adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    /// @notice Enable or disable a feed
    /// @param feed Feed's address
    /// @param isOn Flag setting this feed to enabled or disabled
    function setFeed(address feed, bool isOn) external requiresTrust {
        require(feeds[feed] != isOn, Errors.ExistingValue);
        feeds[feed] = isOn;
        emit FeedChanged(feed, isOn);
    }

    /// @notice Set target's guard
    /// @param target Target address
    /// @param cap The max target that can be deposited on the Divider
    function setGuard(address target, uint256 cap) external requiresTrust {
        guards[target] = cap;
        emit GuardChanged(target, cap);
    }

    /// @notice Set periphery's contract
    /// @param _periphery Target address
    function setPeriphery(address _periphery) external requiresTrust {
        periphery = _periphery;
        emit PeripheryChanged(periphery);
    }
```

Listing 3.3: Example Privileged Opeations in `Divider`

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

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

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.4 Suggested immutable in EmergencyStop/GClaimManager

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `EmergencyStop`, `GClaimManager`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

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."

In the following, we show one key state variable `divider` defined in `EmergencyStop`. If there is no need to dynamically update it after the construction, they can be declared as either constants or `immutable` for gas efficiency. In particular, the state variable `divider` can be defined as `immutable` as it will not be changed after its initialization in `constructor()`.

```
/// @title Stops all feeds from the divider
contract EmergencyStop is Trust {
    address public divider;

    constructor(address _divider) Trust(msg.sender) {
        divider = _divider;
    }
    ...
}
```

Listing 3.4: EmergencyStop.sol

The same suggestion is also applicable to the `GClaimManager` contract.

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

Status The issue has been fixed by this commit: `635aaaa`.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Periphery`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

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 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 3.5: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `Periphery::sponsorSeries()` routine as an example. This routine is designed to trigger default handling. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 28): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function sponsorSeries(
        address feed, uint256 maturity, uint160 sqrtPriceX96
    ) external returns (address zero, address claim) {
        ERC20 stake = ERC20(Feed(feed).stake());
        // transfer stakeSize from sponsor into this contract
        uint256 convertBase = 1;
        uint256 stakeDecimals = stake.decimals();
        if (stakeDecimals != 18) {
            convertBase = stakeDecimals > 18 ? 10 ** (stakeDecimals - 18) : 10 ** (18 -
                stakeDecimals);
        }
        stake.safeTransferFrom(msg.sender, address(this), Feed(feed).stakeSize() /
            convertBase);

        // approve divider to withdraw stake assets
        stake.approve(address(divider), type(uint256).max);

        (zero, claim) = divider.initSeries(feed, maturity, msg.sender);
        address unipool = IUniswapV3Factory(uniFactory).createPool(zero, Feed(feed).
            underlying(), UNI_POOL_FEE); // deploy UNIV3 pool
        IUniswapV3Pool(unipool).initialize(sqrtPriceX96);
        poolManager.addSeries(feed, maturity);
        emit SeriesSponsored(feed, maturity, msg.sender);
    }
```

Listing 3.6: `Periphery::sponsorSeries()`

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

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()`. We highlight that this issue is present in other functions, including `onboardFeed()` from the same contract.

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

Status The issue has been fixed by this commit: `6c3204e`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Sense` protocol, which is a decentralized fixed-income protocol on `Ethereum`, allowing users to manage risk through fixed rates and future yield trading on existing yield bearing-assets. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

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

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

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

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

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

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

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

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