SMART CONTRACT AUDIT REPORT

for

Sodium

Prepared By: Xiaomi Huang

PeckShield

October 17, 2022

Document Properties

Client	Sodium
Title	Smart Contract Audit Report
Target	Sodium
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 17, 2022	Jing Wang	Final Release
1.0-rc	September 16, 2022	Jing Wang	Release Candidate

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 Sodium design document and related smart contract source code, 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 the smart contracts was able to 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 Sodium

Sodium is a hybrid liquidity platform for borrowing against NFT collateral. The protocol allows users to get access to a combination of the instant liquidity market and the peer liquidity market for efficient loan fulfillment and high collateral valuation for a wide variety of whitelisted collections. By combining P2P flexibility with P2Pool efficiency, Sodium is solving what they see to be crucial problems hindering current NFT liquidity platforms. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Sodium

Item	Description
Issuer	Sodium
Website	https://www.sodium.fi/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 17, 2022

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

- https://github.com/sodium-fi/Sodium (8c32434)

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

- https://github.com/sodium-fi/Sodium (05f8b52)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

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

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

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Sodium 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 checklist of 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) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business 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 implementation of the Sodium protocol. 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	3	■ ■ ■
Low	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation was improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key Sodium Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Reentrancy Risk in SodiumCore	Time and State	Partially Fixed
PVE-002	Medium	Potential DoS Against ETHBid()	Business Logic	Fixed
PVE-003	Low	Accommodation of Possible Non-ERC20-Compliance	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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

3 | Detailed Results

3.1 Reentrancy Risk in SodiumCore

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SodiumCore`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `SodiumCore` as an example, the `addFundsERC20()` function (see the code snippet below) is provided to externally call a 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 231) starts before effecting the update on the internal state (lines 240), 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 same entry function.

```
214     function addFundsERC20(
215         uint256 loanId,
216         Types.MetaContribution[] calldata metaContributions,
217         uint256[] calldata amounts,
218         Types.NoWithdrawalSignature calldata noWithdrawalSignature
```

```
    ) external {
        ...
        // Iterate over meta-contributions in order
        for (uint256 i = 0; i < metaContributions.length; i++) {
            _processMetaContribution(
                loanId,
                amounts[i],
                liquidity,
                metaContributions[i]
            );

            // Transfer funds to borrower
            IERC20Upgradeable(currency).transferFrom(
                metaContributions[i].lender,
                borrower,
                amounts[i]
            );

            total += amounts[i];
        }

        loan.liquidity += total;
    }
}
```

Listing 3.1: `SodiumCore::addFundsERC20()`

Note this is a protocol level issue and other routines share the same issue.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been partially fixed by this commit: `2717177`. The `Sodium` team clarifies they have fixed the issue for ETH and non-reentrant ERC20s, as this is what they plan to support at launch. They explained that preventing ERC20 re-entrancy would add unnecessary gas to ERC20-loan-related functions.

3.2 Potential DoS Against ETHBid()

- ID: PVE-002
- Severity: Medium
- Likelihood: low
- Impact: Medium
- Target: `SodiumCore`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `SodiumCore` contract provides an `ETHBid()` routine for users to make an `ETH` bid in a collateral auction. While examining the current `ETHBid()` logic, we notice the existence of potential DoS (denial-of-service) that needs to be avoided in the implementation.

To elaborate, we show below the implementation of the `ETHBid()` routine. It will repay the `rawBid` price of `ETH` to the previous bidder. However, it comes to our attention that the `ETHBid()` routine may always revert if the previous bidder refuses to receive `ETH`. As a result, the previous bidder will finally win and withdraw the `NFT` after the `ETHBid()`.

```
    function ETHBid(uint256 id, uint256 index)
        external
        payable
        nonReentrant
        duringAuctionOnly(id)
    {
        require(loans[id].currency == address(0), "3");

        address bidder = auctions[id].bidder;

        // Repay previous bidder if needed
        if (bidder != address(0)) {
            payable(bidder).transfer(auctions[id].rawBid);
        }

        _executeBid(id, msg.value, index);
    }
```

Listing 3.2: `SodiumCore::ETHBid()`

Recommendation Avoid the above denial-of-service risk in the above `ETHBid()` routines.

Status The issue has been fixed by this commit: `2717177`.

3.3 Accommodation of Possible Non-ERC20-Compliance

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Sodium`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transferFrom()` 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. Specifically, the `transferFrom()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transferFrom()` interface with a `bool` return value. As a result, the call to `transferFrom()` may expect a return value. With the lack of return value of `USDT`'s `transferFrom()`, the call will be unfortunately reverted.

```
    function transferFrom(address _from, address _to, uint _value) public
        onlyPayloadSize(3 * 32) {
        var _allowance = allowed[_from][msg.sender];

        // Check is not needed because sub(_allowance, _value) will already throw if
            this condition is not met
        // if (_value > _allowance) throw;

        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        if (_allowance < MAX_UINT) {
            allowed[_from][msg.sender] = _allowance.sub(_value);
        }
        uint sendAmount = _value.sub(fee);
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(_from, owner, fee);
        }
        Transfer(_from, _to, sendAmount);
    }
```

Listing 3.3: USDT Token Contract

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, 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()`.

In current implementation, if we examine the `Sodium::_executeRepayment()` routine that is designed to add liquidity to the pool with the given amounts of tokens. To accommodate the specific idiosyncrasy, there is a need to use `safeTransferFrom()`, instead of `transferFrom()` (line 707).

```
697     function _executeRepayment(
698         uint256 loanId,
699         uint256 available,
700         address currency,
701         address from
702     ) internal {
```

```
        ...
        uint256 amount = principal + interest;

        // Repay lender
        IERC20Upgradeable(currency).transferFrom(from, lender, amount);

        // Send fee
        IERC20Upgradeable(currency).transferFrom(from, sodiumTreasury, fee);

        // Decreasing amount of funds available for further repayment
        available -= amount + fee;

        emit RepaymentMade(loanId, lender, amount);
        ...
    }
```

Listing 3.4: Sodium::_executeRepayment()

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

Status The issue has been fixed by this commit: 2717177.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Sodium
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the Sodium protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration). 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 owner account and its related privileged accesses in current contract.

To elaborate, we show the setWalletFactory() routine from the Sodium contract. This function allows the owner account set the address of factory which could create user wallet to hold NFT.

```
    function setWalletFactory(address factory) external onlyOwner {
        sodiumWalletFactory = ISodiumWalletFactory(factory);
    }
```

Listing 3.5: Sodium::setWalletFactory()

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. 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.

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. The team clarifies they plan on using a multi-sig wallet to start, and eventually migrating ownership of sensitive contracts to timelock plus `DAO`-like governance contract.

4 Conclusion

In this audit, we have analyzed the `Sodium` design and implementation. `Sodium` is a hybrid liquidity platform for borrowing against `NFT` collateral. The current code base is well structured and neatly organized. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

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