SMART CONTRACT AUDIT REPORT

for

SLOHM FINANCE

Prepared By: Xiaomi Huang

PeckShield

May 7, 2026

Document Properties

Client	Slohm Finance
Title	Smart Contract Audit Report
Target	Slohm Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 7, 2026	Xuxian Jiang	Final Release
1.0-rc	May 2, 2026	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Slohm Finance 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 Slohm Finance

Slohm Finance is an OPNet implementation of a treasury-backed reserve currency with a rebasing staking receipt and voucher-authorized bond issuance. Physical reserves live outside the contracts; on-chain safety depends on the Treasury TVL feed, the floor-price accounting invariant, strict SLOHM mint authority, and correct BondDepository reservation/redemption accounting. The basic information of the Slohm Finance protocol is as follows:

Table 1.1: Basic Information of The Slohm Finance Protocol

Item	Description
Issuer	Slohm Finance
Type	OPNet Smart Contract
Platform	AssemblyScript
Audit Method	Whitebox
Latest Audit Report	May 7, 2026

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

- https://github.com/Slohm-fi/Slohm-contracts.git (e844483)

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

- https://github.com/Slohm-fi/Slohm-contracts.git (fdd71d6, 438b823)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

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

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	OP20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], 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 `Slohm Finance` 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	3
Low	4
Informational	1
Total	8

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 3 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Slohm Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Missing Max Supply Enforcement in yS-LOHM::_mint()	Business Logic	Resolved
PVE-002	Medium	Revisited Cooldown Use For Stake Withdrawal	Business Logic	Resolved
PVE-003	Low	Possible Stake Frontrunning For Rebase Profit	Time And State	Resolved
PVE-004	Medium	Tightened Caller Validation of yS-LOHM::rebase()	Security Features	Resolved
PVE-005	Low	Improved setPaused() Logic in Staking/Treasury/BondDepository	Coding Practices	Resolved
PVE-006	Informational	Revisited Event Emission in SLOHM/yS-LOHM	Coding Practices	Resolved
PVE-007	Low	Non-initialized Return Values in SlohmAuthority	Coding Practices	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Resolved

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 Missing Max Supply Enforcement in ySLOHM::_mint()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ySLOHM
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the Slohm Finance protocol, there is a core ySLOHM contract, an OP-20 token contract that acts as a rebasing staking receipt with gons-per-fragment accounting. Specifically, the staking component of the protocol allows participants to stake SLOHM tokens and get ySLOHM in return. ySLOHM is a rebasing, OP-20 token that evenly distributes profits to staking users, i.e., ySLOHM holders. While examining the token contract, we notice an issue in not enforcing the maximum supply that can ever be minted.

To elaborate, we show below the _mint() implementation in the ySLOHM token contract. This function has a rather straightforward logic in updating the total supply and the total gons tracked as well as allocating new gons to the intended recipient. However, we notice current logic does not check against the maximum supply that can ever be minted. Be design, the maximum supply amount is 1B * 10^18, which needs to honored when new ySLOHM tokens are minted. Note this issue also affects another routine in the same contract, i.e., rebase().

```
169     protected override _mint(to: Address, amount: u256): void {
170         if (to === Address.zero()) {
171             throw new Revert('ySLOHM: mint to zero address');
172         }
174         const gonsPerFrag: u256 = this._gonsPerFragment.value;
175         const gonValue: u256 = SafeMath.mul(amount, gonsPerFrag);
177         // Update total supply
178         const currentSupply: u256 = this._totalSupply.value;
179         this._totalSupply.value = SafeMath.add(currentSupply, amount);
```

```
181         // Update total gons tracked
182         const currentTotalGons: u256 = this._totalGons.value;
183         this._totalGons.value = SafeMath.add(currentTotalGons, gonValue);

185         // Allocate gons to recipient
186         const toGons: u256 = this._gonBalances.get(to);
187         this._gonBalances.set(to, SafeMath.add(toGons, gonValue));

189         this.createTransferredEvent(Blockchain.tx.sender, Address.zero(), to, amount);
190     }
```

Listing 3.1: `ySLOHM::_mint()`

Recommendation Revise the above-mentioned routines (`_mint()` and `rebase()`) to ensure the total supply of `ySLOHM` tokens do not exceed the specified maximum supply of `1B * 10^18`.

Status The issue has been resolved by the following commits: `1c632f7` and `7b3eefd`.

3.2 Revisited Cooldown Use For Stake Withdrawal

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier, `Slohm Finance` is in essence a treasury-backed reserve currency with a rebasing staking receipt. The stake/unstake mechanics are provided in `Staking` with required cooldown period for withdrawal settlement. In the process of examining the stake withdrawal logic, we notice it makes use of latest cooldown period, instead of the cooldown period at the time when the withdraw is requested.

To elaborate, we show below the implementation of the related `withdraw()` routine. As the name indicates, it settles the withdrawal request after the cooldown period is expired. However, it comes to our attention that the cooldown period being used for expiry validation is the one currently in effect (lines 373-378), not the one when the request is made. As a result, a withdrawing staker may be blocked if a large cooldown period is applied after the withdrawal request is made.

```
359     @method
360     @emit('WithdrawCompleteEvent')
361     public withdraw(calldata: Calldata): BytesWriter {
362         this.requireNotPaused();
```

```
363         this.nonReentrant();
364
365         const sender = Blockchain.tx.sender;
366
367         const pendingAmount = this._withdrawAmount.get(sender);
368         if (pendingAmount.isZero()) {
369             throw new Revert('Staking: no pending withdrawal');
370         }
371
372         const startBlock = this._withdrawStartBlock.get(sender);
373         const cooldown = this._cooldownPeriod.value;
374         const requiredBlock = SafeMath.add(startBlock, cooldown);
375
376         if (Blockchain.block.numberU256 < requiredBlock) {
377             throw new Revert('Staking: cooldown not expired');
378         }
379
380         // Clear withdrawal request
381         this._withdrawAmount.set(sender, u256.Zero);
382         this._withdrawStartBlock.set(sender, u256.Zero);
383
384         // Transfer sLOHM back to user
385         const slohmAddr = this._slohmToken.value;
386         TransferHelper.transfer(slohmAddr, sender, pendingAmount);
387
388         this.emitEvent(new WithdrawCompleteEvent(sender, pendingAmount));
389
390         this._locked.value = false;
391         return new BytesWriter(0);
392     }
```

Listing 3.2: Staking::withdraw()

Recommendation Revisit the above logic to make use of the intended cooldown period when finalizing user withdrawal requests.

Status The issue has been resolved by this commit: `6e31e18`.

3.3 Possible Stake Frontrunning For Rebase Profit

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Treasury`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

As mentioned earlier, the `Slohm Finance` protocol implements a treasury-backed reserve currency with a rebasing staking receipt and voucher-authorized bond issuance. While evaluating the rebasing mechanism, we notice it is triggered for each epoch based on the adjustable reward rate for `ySLOHM` holders. With that, we observe a frontrunning opportunity where a user may stake right before the rebase is triggered. Fortunately, the staker still needs to go through the withdraw cooldown.

To elaborate, we show below the implementation of the `rebase()` routine. This routine is part of the `Treasury` contract and is used to increase the `ySLOHM`'s `totalSupply` and inflate all holders' balances proportionally. With that, if a user stakes `SLOHM` for `ySLOHM` right before the `rebase()` routine is executed, the user may enjoy the benefit preferably when compared with earlier stakers. As mentioned earlier, all stakers need to wait for the cooldown period before their withdraws. From another perspective, the `OPNet` does not protect against cross-block intra-epoch frontrunning, which relaxes the constraints for frontrunning.

```
1507        for (let i: i32 = 0; i < 10; i++) {
1508            epochEnd = this._epochEnd.value;
1509            if (currentBlock < epochEnd) break;

1511            const epochNum = this._epochNumber.value;

1513            // Graceful degradation: stale TVL -> skip mint, still advance epoch
1514            if (this.isTvlStale()) {
1515                this._epochNumber.value = SafeMath.add(epochNum, u256.One);
1516                this._epochEnd.value = SafeMath.add(epochEnd, epochLength);
1517                this.emitEvent(new RebaseSkipped(epochNum, Treasury.SKIP_STALE));
1518                this.adjust();
1519                continue;
1520            }

1522            // No stakers -> skip mint (S2-03). Minting to Staking when ySLOHM
1523            // supply is zero creates unallocated sLOHM that no receipt holder
1524            // can claim, and silently burns Treasury reserves.
1525            if (this.yslohmTotalSupply().isZero()) {
1526                this._epochNumber.value = SafeMath.add(epochNum, u256.One);
1527                this._epochEnd.value = SafeMath.add(epochEnd, epochLength);
1528                this.emitEvent(new RebaseSkipped(epochNum, Treasury.SKIP_NO_STAKERS));
```

```
1529            this.adjust();
1530            continue;
1531        }

1533        // Compute requested reward = _rewardRate x _trackedSlohmSupply / 1e6
1534        const rate = this._rewardRate.value;
1535        const supply = this._trackedSlohmSupply.value;
1536        let requested = u256.Zero;
1537        if (!supply.isZero() && !rate.isZero()) {
1538            requested = SafeMath.div(SafeMath.mul(supply, rate), RATE_DENOM);
1539        }

1541        // Cap at excess
1542        const excess = this.computeExcess();
1543        let actual = requested;
1544        if (actual > excess) {
1545            actual = excess;
1546        }

1548        // Mint + rebase if non-zero
1549        if (!actual.isZero()) {
1550            this.slohmMint(stakingAddr, actual);
1551            this.yslohmRebase(epochNum, actual);
1552            totalMinted = SafeMath.add(totalMinted, actual);
1553            this.emitEvent(new RebaseExecuted(epochNum, actual));
1554        } else {
1555            this.emitEvent(new RebaseSkipped(epochNum, Treasury.SKIP_ZERO_EXCESS));
1556        }

1558        // Advance epoch
1559        this._epochNumber.value = SafeMath.add(epochNum, u256.One);
1560        this._epochEnd.value = SafeMath.add(epochEnd, epochLength);

1562        // Step rate toward target
1563        this.adjust();
1564    }
```

Listing 3.3: `Treasury::rebase()`

Note that this is a common sandwich-based arbitrage behavior plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user. We need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above frontrunning behavior to better protect the rebasing operation in `Slohm`.

Status The issue has been resolved with the built-in cooldown period for user withdrawals.

3.4 Tightened Caller Validation of ySLOHM::rebase()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: `ySLOHM`
- Category: Security Features [7]
- CWE subcategory: CWE-282 [2]

Description

As mentioned in Section 3.3, the `Slohm Treasury` is expected to trigger the rebase mechanism for each epoch to inflate all `ySLOHM` holders' balances proportionally. Each rebase mints the excess-capped reward amount based on the reward rate and tracked supply. In the analysis of the rebase mechanism, we notice it should be callable only from the `Treasury` contract. However, current implementation allow both `Treasury` and `Staking` to trigger the rebase.

To elaborate, we show below the `rebase()` implementation from the `ySLOHM` token contract. We notice the presence of an internal helper routine `onlyStakingOrRebaseAuthority()` that validates the caller to be `Staking` or `Treasury`. As analyzed earlier, we can further restrict the caller to be the `Treasury` contract only.

```
720     public rebase(calldata: Calldata): BytesWriter {
721         const epoch: u256 = calldata.readU256();
722         const profit: u256 = calldata.readU256();

724         this.onlyStakingOrRebaseAuthority();

726         const currentSupply: u256 = this._totalSupply.value;
727         const totalGons: u256 = this._totalGons.value;

729         // If no supply or no profit, nothing to rebase
730         if (currentSupply.isZero()  profit.isZero()) {
731             const response = new BytesWriter(32);
732             response.writeU256(currentSupply);
733             return response;
734         }

736         // New total supply
737         const newTotalSupply: u256 = SafeMath.add(currentSupply, profit);
738         this._totalSupply.value = newTotalSupply;

740         // Recalculate gonsPerFragment: totalGons / newTotalSupply
741         // This decrease in gonsPerFragment means each gon is worth more ySLOHM
742         if (!totalGons.isZero()) {
743             const newGonsPerFragment = SafeMath.div(totalGons, newTotalSupply);
744             if (newGonsPerFragment.isZero()) {
745                 throw new Revert('ySLOHM: gonsPerFragment would reach zero - rebase
                        rejected');
```

```
746             }
747             this._gonsPerFragment.value = newGonsPerFragment;
748         }
749
750         // Update currentIndex: index = index * newTotalSupply / oldTotalSupply
751         const oldIndex: u256 = this._currentIndex.value;
752         const newIndex: u256 = SafeMath.div(SafeMath.mul(oldIndex, newTotalSupply),
                currentSupply);
753         this._currentIndex.value = newIndex;
754
755         // Emit rebase event
756         this.emitEvent(new RebaseEvent(epoch, profit, newTotalSupply, newIndex));
757
758         const response = new BytesWriter(32);
759         response.writeU256(newTotalSupply);
760         return response;
761     }
```

Listing 3.4: `ySLOHM::rebase()`

Recommendation Revise the `onlyStakingOrRebaseAuthority()` routine to ensure the caller is `Treasury` only.

Status The issue has been resolved by the following commits: `b06eaff` and `f8a8bac`.

3.5 Improved setPaused() Logic in Staking/Treasury/BondDepository

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`, `Treasury`, `BondDepository`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [1]

Description

The `Slohm Finance` protocol has four upgradeable contracts, i.e., `Staking`, `BondDepository`, `Treasury`, and `SlohmAuthority`. Each contract allows the privileged admin to pause or upgrade. While reviewing the pause logic, we notice current implementation may be improved.

To elaborate, we use the `Staking` contract as an example and show below the implementation of its `setPaused()` routine. The pause control has dual-authority, i.e., `governor` or `_pauseAuthority`. And there are two internal variables `isGovernor` and `isPauseAuth` to represent whether the caller is `governor` or `_pauseAuthority`, respectively. With the dual-authority, we can optimize the internal `isGovernor`

logic from the `if`-condition (line 616) on `authorityAddr !== Address.zero()` to be `!isPauseAuth && authorityAddr !== Address.zero())`. By doing so, we only need to evaluate the `isGovernor` variable when `isPauseAuth` is false. Note the same improvement can be applied to other `setPaused()` routines in `Treasury` and `BondDepository` contracts as well.

```
609     public setPaused(calldata: Calldata): BytesWriter {
610         const sender = Blockchain.tx.sender;
611         const authorityAddr = this._authority.value;
612         const pauseAuth = this._pauseAuthority.value;
613
614         const isPauseAuth = pauseAuth !== Address.zero() && sender === pauseAuth;
615         let isGovernor = false;
616         if (authorityAddr !== Address.zero()) {
617             if (sender === authorityAddr) {
618                 isGovernor = true;
619             } else if (!isPauseAuth) {
620                 const cd = new BytesWriter(4);
621                 cd.writeSelector(encodeSelector('governor()'));
622                 const result = Blockchain.call(authorityAddr, cd);
623                 if (!result.success) {
624                     throw new Revert('Staking: authority call failed');
625                 }
626                 const governor: Address = result.data.readAddress();
627                 isGovernor = sender === governor;
628             }
629         }
630
631         if (!isGovernor && !isPauseAuth) {
632             throw new Revert('Staking: not authorized to pause');
633         }
634
635         const paused = calldata.readBoolean();
636         this._paused.value = paused;
637
638         if (paused) {
639             this.emitEvent(new Paused());
640         } else {
641             this.emitEvent(new Unpaused());
642         }
643
644         return new BytesWriter(0);
645     }
```

Listing 3.5: `Staking::setPaused()`

Recommendation Optimize the above-mentioned `setPaused()` routines by avoiding redundant computation.

Status The issue has been resolved by this commit: `eb82ade`.

3.6 Revisited Event Emission in ySLOHM

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ySLOHM
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

In OPNET, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the ySLOHM contract as an example. It is a rebasing staking receipt token (OP20, 18 decimals) whose balance grows automatically each epoch. And the token contract implementation is inherited from the OP20 contract. In the following, we show the _mint() implementation from OP20 and ySLOHM token contracts, respectively.

The comparison is helpful to expose the difference of emitted events in the common _mint() function. In particular, the _mint() function from the reference OP20 contract emits the OP20MintedEvent event while the _mint() function from ySLOHM emits another OP20TransferredEvent event. For consistency, it is helpful to also emit the OP20MintedEvent event along with current OP20TransferredEvent event in ySLOHM's _mint() function. Similarly, the _burn() function in ySLOHM can also be improved by additionally emitting the OP20BurnedEvent event.

```
913     protected _mint(to: Address, amount: u256): void {
914         if (to === Address.zero()) {
915             throw new Revert('Invalid receiver');
916         }

918         const toBal: u256 = this.balanceOfMap.get(to);
919         this.balanceOfMap.set(to, SafeMath.add(toBal, amount));

921         // @ts-expect-error AssemblyScript valid
922         this._totalSupply += amount;

924         if (this._totalSupply.value > this._maxSupply.value) {
925             throw new Revert('Max supply reached');
926         }

928         this.createMintedEvent(to, amount);
```

```
929     }
```

Listing 3.6: `OP20::_mint()`

```
169     protected override _mint(to: Address, amount: u256): void {
170         if (to === Address.zero()) {
171             throw new Revert('ySLOHM: mint to zero address');
172         }
174         const gonsPerFrag: u256 = this._gonsPerFragment.value;
175         const gonValue: u256 = SafeMath.mul(amount, gonsPerFrag);
177         // Update total supply
178         const currentSupply: u256 = this._totalSupply.value;
179         this._totalSupply.value = SafeMath.add(currentSupply, amount);
181         // Update total gons tracked
182         const currentTotalGons: u256 = this._totalGons.value;
183         this._totalGons.value = SafeMath.add(currentTotalGons, gonValue);
185         // Allocate gons to recipient
186         const toGons: u256 = this._gonBalances.get(to);
187         this._gonBalances.set(to, SafeMath.add(toGons, gonValue));
189         this.createTransferredEvent(Blockchain.tx.sender, Address.zero(), to, amount);
190     }
```

Listing 3.7: `ySLOHM::_mint()`

Recommendation Revise the above routines to additionally emit the `OP20MintedEvent` event and the `OP20BurnedEvent` event in `_mint()` and `_burn()` respectively.

Status The issue has been resolved by the following commits: `50f8747` and `7ff33d8`.

3.7 Non-initialized Return Values in SlohmAuthority

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SlohmAuthority`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

In `OPNET`, the governance model is largely managed in the `SlohmAuthority` contract. While examining the governance logic, we notice a number of administrative functions have return values that are not initialized.

To elaborate, we show below the `cancelPushGovernor()` function as an example. It comes to our attention that this function returns `new BytesWriter(1);` with one byte of non-initialized value. To avoid confusion to off-chain indexers or analytic engines, we suggest to return a fixed value of `0x1` or a `true` `boolean` value. Note this issue affects a number of routines, including `pushGovernor()`, `executePushGovernor()`, `cancelPushGovernor()`, `pullGovernor()`, `pushVault()`, `executePushVault()`, `cancelPushVault()`, `pushGuardian()`, `pullGuardian()`, `pushPolicy()`, `pullPolicy()`, `addAdmin()`, `removeAdmin()`, `setPauseTargetTreasury()`, `setPauseTargetBondDepository()`, `setPauseTargetStaking()`, `pause()`, `unpause()`, `pauseAll()`, and `unpauseAll()`.

```
233     public cancelPushGovernor(calldata: Calldata): BytesWriter {
234         this.onlyGuardian();
235         if (this._governorQueuedAt.value.isZero()) {
236             throw new Revert('SlohmAuthority: no pending governor change');
237         }
238         const cancelled = this._pendingGovernorTL.value;
239         this._pendingGovernorTL.value = Address.zero();
240         this._governorQueuedAt.value = u256.Zero;
241         this.emitEvent(new GovernorQueueCancelled(cancelled));
242         return new BytesWriter(1);
243     }
```

Listing 3.8: `SlohmAuthority::cancelPushGovernor()`

Recommendation Revise the above-mentioned routines to properly return unambiguous return value.

Status The issue has been resolved by the following commit: `f3c293b`.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Slohm Finance` protocol, there is a privileged admin account that plays a critical role in governing the `SlohmAuthority` contract that holds privileged access over the rest of protocol contracts. In the following, we show representative privileged operations in the `Slohm Finance` protocol.

```
246     @method({ name: 'newAddr', type: ABIDataTypes.ADDRESS })
247     @emit('GuardianPushed')
248     public pushGuardian(calldata: Calldata): BytesWriter {...}
249     ...
250     @method({ name: 'newAddr', type: ABIDataTypes.ADDRESS })
251     @emit('PolicyPushed')
252     public pushPolicy(calldata: Calldata): BytesWriter {...}
253     ...
254     @method
255     @emit('Paused')
256     public pause(calldata: Calldata): BytesWriter {...}
257     ...
258     @method
259     @emit('Unpaused')
260     public unpause(calldata: Calldata): BytesWriter {...}
261     ...
262     @method
263     @emit('AllPaused')
264     public pauseAll(calldata: Calldata): BytesWriter {...}
265     ...
266     @method
267     @emit('AllUnpaused')
268     public unpauseAll(calldata: Calldata): BytesWriter {...}
```

Listing 3.9: Example Privileged Operations in `SlohmAuthority`

```
460     @method({ name: 'addr', type: ABIDataTypes.ADDRESS })
461     @emit('SlohmTokenUpdated')
462     public setSlohmToken(calldata: Calldata): BytesWriter {...}

464     @method({ name: 'addr', type: ABIDataTypes.ADDRESS })
465     @emit('YslohmTokenUpdated')
466     public setYslohmToken(calldata: Calldata): BytesWriter {...}

468     @method({ name: 'addr', type: ABIDataTypes.ADDRESS })
```

```
469     @emit('StakingContractUpdated')
470     public setStakingContract(calldata: Calldata): BytesWriter {...}
471
472     @method({ name: 'addr', type: ABIDataTypes.ADDRESS })
473     @emit('BondDepositoryUpdated')
474     public setBondDepository(calldata: Calldata): BytesWriter {...}
475
476     @method({ name: 'addr', type: ABIDataTypes.ADDRESS })
477     @emit('TvlUpdaterUpdated')
478     public setTvlUpdater(calldata: Calldata): BytesWriter {...}
479
480     @method({ name: 'addr', type: ABIDataTypes.ADDRESS })
481     @emit('PauseAuthorityUpdated')
482     public setPauseAuthority(calldata: Calldata): BytesWriter {...}
483
484     @method(
485         { name: 'permission', type: ABIDataTypes.UINT256 },
486         { name: 'addr', type: ABIDataTypes.ADDRESS },
487     )
488     @emit('PermissionGranted')
489     @emit('PermissionGrantQueued')
490     public enable(calldata: Calldata): BytesWriter {...}
```

Listing 3.10: Example Privileged Operations in `Treasury`

We emphasize that the privilege assignment with various protocol contracts is necessary and required for proper protocol operations. However, it is worrisome if the `admin` account is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account. And a number of security-sensitive operations are behind a timelock. Note that a compromised `admin` account would allow the attacker to change current `governor`/`guardian`, which holds privileged accesses to other protocol contracts, including `Treasury`, `Staking`, `BondNotes`, `BondDepository`, `SLOHM`, and `ySLOHM`.

Moreover, it should be noted that current contracts have the support of being upgraded after the deployment. And there is a need to properly manage the deployment key for upgrade as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended governance contract. All changes from privileged operations may need to be mediated with necessary timelocks.

Status This issue has been resolved as changes from privileged operations have been mediated with use of necessary timelocks.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Slohm Finance protocol, which is an OPNet implementation of a treasury-backed reserve currency with a rebasing staking receipt and voucher-authorized bond issuance. Physical reserves live outside the contracts; on-chain safety depends on the Treasury TVL feed, the floor-price accounting invariant, strict SLOHM mint authority, and correct BondDepository reservation/redemption accounting. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that AssemblyScript-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-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

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

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

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

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

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

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

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

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

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

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

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