Persistence

Security Assessment

April 26, 2022

Prepared for:

Puneet Mahajan

Persistence

Prepared by:

Devashish Tomar and David Pokora

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Persistence engaged Trail of Bits to review the security of its ETH 2.0 liquid staking smart contracts. From February 22 to March 7, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and relevant documentation provided by the Persistence team.

Summary of Findings

The audit uncovered two significant flaws that could impact system confidentiality, integrity, or availability. These issues concern the deployment of the system and a sandwich attack risk that arises when a user's stake is slashed. We set the severity of two of the issues, which concern value inconsistencies, to undetermined.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Low	1
Informational	2
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	2
Configuration	2
Data Validation	2
Timing	2

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

David Pokora, Consultant
david.pokora@trailofbits.com

Devashish Tomar, Consultant
devashish.tomar@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 17, 2022	Pre-project kickoff call
February 28, 2022	Status update meeting #1
March 7, 2022	Delivery of report draft and report readout meeting
April 6, 2022	Delivery of final report with fix log
April 26, 2022	Delivery of public report and fix log

Project Goals

The engagement was scoped to provide a security assessment of Persistence's ETH 2.0 liquid staking smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is the arithmetic and accounting related to the ETH 2.0 token deposit functions sound?
- Are access controls used appropriately, and are the access controls for all roles (such as the node operator role) enforced appropriately?
- Is the low-level code generally sound, and does it use pointer arithmetic and data validation correctly?
- Is the state machine appropriate for the ETH 2.0 liquid staking contracts? Are there any unconsidered edge cases that could lead to undefined behavior?
- Are validators registered properly? Could a validator be re-registered or associated with an incorrect public key?
- Is the system susceptible to front-running attacks?
- Do critical operations trigger events that would be sufficient to form an audit trail in the event of an issue or attack?
- Could the data validation performed by the contracts result in a blocked state transition?

Project Targets

The engagement involved a review and testing of the following target.

`eth2-liquid-staking-contracts`

Repository	https://github.com/persistenceOne/eth2-liquid-staking-contracts
Version	`100b879cbb53f3925ee4b4c173acaf38ba2254cb`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- Analysis of the access controls throughout the codebase did not reveal any concerns.
- A review of the system's data validation found that the return values of several ERC20 token functions are not checked (TOB-PER2-1); similarly, various setter functions in the contracts lack zero-address checks (TOB-PER2-2).
- A check of the contracts for front-running opportunities revealed that the `Core` contract's initialization method is vulnerable to front-running (TOB-PER2-3).
- Analysis of critical functions' emission of events identified a number of functions that do not emit events for state-changing operations (TOB-PER2-5) and found that one event is emitted unintuitively (TOB-PER2-4).
- A review of the arithmetic throughout the codebase revealed inconsistencies in the `Issuer` and `KeysManager` contracts' definitions of `VALIDATOR_DEPOSIT` (TOB-PER2-6).
- Analysis of the low-level code and its correctness found that it is used appropriately and correctly implements pointer arithmetic.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive dynamic testing of the ETH 2.0 liquid staking contracts. Although we received a testing environment during the last week of the assessment, the need to triage outstanding concerns through a manual review of the smart contracts prevented us from performing comprehensive dynamic testing.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	A review of the arithmetic identified inconsistent definitions of `VALIDATOR_DEPOSIT`. The codebase also uses magic numbers in multiple places, and the system would benefit from additional documentation on its arithmetic.	Moderate
Authentication / Access Controls	The smart contracts' access controls are sufficient, and all roles are enforced appropriately. However, the roles should be explicitly documented and would benefit from unit testing.	Satisfactory
Complexity Management	The smart contracts are not overly complex, and their functionality is generally fairly simple and easy to understand. However, we identified many code quality issues (appendix C) and found that additional code comments would clarify the intention behind parts of the codebase.	Moderate
Decentralization	Operators hold a unique role in the system and can perform operations that affect end users. For instance, `Oracle` governors and node operators can choose to censor a validator. Moreover, they can set the values of limits and contract references; if those values are unintuitive, the system may not function as intended.	Moderate
Documentation	The documentation provided by the Persistence team outlines the overarching design philosophy of the smart contracts. However, additional operational and user documentation would be beneficial, and further clarification on the system's design would be helpful to external contributors and auditors.	Moderate

Front-Running Resistance	The initialization of the `Core` contract is vulnerable to front-running, although front-running would cause the contract's deployer to incur only a minor loss of funds. We did not identify any other front-running issues; however, we did not exhaustively check the protocol for front-running opportunities.	**Further Investigation Required**
Low-Level Manipulation	The low-level calls in the system are generally appropriate. A review of the pointer arithmetic and low-level code's correctness did not identify any concerns.	**Strong**
Testing and Verification	Although the Persistence team has a local testing environment in which it can perform some integration tests, the test suite is largely incomplete and does not cover various cases. Additionally, the process of preparing and running the tests is nontrivial and may be prone to developer error.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Missing checks of common ERC20 functions' return values	Data Validation	**Medium**
2	Lack of checks for the zero address	Data Validation	**Low**
3	Core's init method is prone to front-running	Timing	**High**
4	Emission of duplicate events in a loop	Auditing and Logging	**Informational**
5	Lack of events for critical operations	Auditing and Logging	**Informational**
6	Inconsistent VALIDATOR_DEPOSIT values	Configuration	**Undetermined**
7	StakingPool.slash function is vulnerable to sandwich attacks	Timing	**High**
8	Erroneous state variable shadowing	Configuration	**Undetermined**

Detailed Findings

1. Missing checks of common ERC20 functions' return values	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PER2-1
Targets: `contracts/Oracle.sol`, `contracts/StakingPool.sol`	

Description

The `StkEth` token contract exposes several ERC20 token functions that revert upon a failure or provide a return value to indicate whether an operation succeeded. However, the callers of these functions do not check their return values.

These callers include the `StakingPool` contract, which does not check the return values of the `approve`, `transfer`, and `transferFrom` methods:

```
function updateRewardPerValidator(uint256 newReward) public override {

    uint256 totalValidators = IOracle(core.oracle()).activatedValidators() +
IIssuer(core.issuer()).pendingValidators();

    stkEth.transferFrom(_msgSender(), address(this), newReward);
```

Figure 1.1: `contracts/StakingPool.sol#L49-L53`

Similarly, the `Oracle` contract does not check the results of its calls to `StkEth`'s `approve` method:

```
stkEth().approve(core().validatorPool(), type(uint256).max);
```

Figure 1.2: `contracts/Oracle.sol#L102`

Exploit Scenario

The `StkEth` token contract is updated such that its `approve`, `transfer`, and `transferFrom` functions return `false` instead of reverting when an operation fails. However, because the `Oracle` and `StakingPool` contracts do not check the functions' return values, calls to those functions result in a loss of funds or undefined behavior.

Recommendations

Short term, add validation of the results of calls to the `approve`, `transfer`, and `transferFrom` functions. That way, if the code is changed such that the functions can return `false`, failed transactions will revert. Additionally, consider using OpenZeppelin's `SafeERC20` library in interactions with any external ERC20 tokens.

Long term, adhere to the token integration best practices outlined in appendix D. Ensure that there are appropriate checks of all return values. Lastly, consider using static analyzers such as Slither to catch missing return value checks.

2. Lack of checks for the zero address

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PER2-2
Target: `contracts/Core.sol`, `contracts/CoreRef.sol`, `contracts/Issuer.sol`, `contracts/StakingPool.sol`	

Description

The constructor of the `StakingPool` contract does not check the `_weth` argument, which defines the wrapped ether token contract, against the zero address. As a result, the `StakingPool` contract may be deployed with a `_weth` argument that has been initialized incorrectly.

```
function initialize (IERC20 _pstake, IUniswapRouter _router, ICore _core, address
_weth)
    public initializer
{
    __Ownable_init();
    pstake = _pstake;
    core = _core;
    stkEth = core.stkEth();
    router = _router;
    WETH = _weth;
```

Figure 2.1: `contracts/StakingPool.sol#L37-L45`

Other setter functions also fail to validate the addresses they receive as input. The following addresses are not validated:

- Those passed to the `initialize` function of the `StakingPool` contract
- The `core` address passed to the constructor of the `CoreRef` contract
- The `demoDeposit` address passed to the constructor of the `Issuer` contract
- The `_address` argument passed to the `set` function of the `Core` contract

Exploit Scenario

An operator uses an off-chain tool to deploy the `StakingPool` smart contract. However, because of a bug, the software fails to initialize the `_weth` argument, leaving its address as the zero address. The constructor does not detect the use of the zero address, and the contract is deployed. The operator believes the deployment to have been successful, but it leads to undefined behavior.

Recommendations

Short term, add zero-value or contract existence checks to the functions listed above to ensure that users cannot accidentally set their arguments to the zero address, misconfiguring the protocol.

Long term, integrate Slither into the development process to identify missing zero-address checks.

3. Core’s init method is prone to front-running	
Severity: **High**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-PER2-3
Target: `contracts/Core.sol`	

Description

The deployer of the `Core` contract is expected to call the `init` function immediately after the contract’s deployment to set him- or herself as the contract’s governor and to create a new `StkEth` token instance. However, because the `init` method must be called after deployment, an attacker could front-run the call to set him- or herself as the contract’s governor instead.

```
function init() external override initializer {

    _setupGovernor(msg.sender);

    StkEth _stkEth = new StkEth(address(this));
    _setStkEth(address(_stkEth));
}
```

Figure 3.1: `contracts/Core.sol#L29-L35`

Exploit Scenario

An operator deploys the `Core` contract and then sends a transaction calling the `init` function to become the contract’s governor. However, an attacker notices the pending transaction and sends another transaction to the `init` function, paying a higher gas fee to ensure that his transaction will be accepted first. As a result, the attacker is set as the contract’s governor.

Recommendations

Short term, add a call to the `init` method from the constructor of the `Core` contract. This will ensure that the governor is set immediately upon the contract’s deployment and will prevent front-running of calls to the `init` method.

Long term, implement access controls on all sensitive methods used to assign roles in the system; this will ensure that those methods cannot be front-run by malicious parties. Additionally, identify any upgradeable contracts and use a constructor to initialize the state of non-upgradeable contracts.

4. Emission of duplicate events in a loop

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-PER2-4
Target: `contracts/KeysManager.sol`	

Description

The `KeysManager` contract provides a method, `activateValidator`, that the `Oracle` calls to activate validators (by activating their public keys). The method loops through a list of public keys and emits an `ActivateValidator` event each time it activates a key. However, with every call, it emits the same event—an event listing all of the public keys activated in the call. The event may be somewhat useful, since it indicates how many validators were activated in a transaction; however, events with parameters distinguishing between the public keys that were activated would provide a more useful audit trail.

```
for (uint256 i = 0; i < publicKeys.length; i++) {
    Validator storage validator = _validators[publicKeys[i]];
    if (validator.state == State.ACTIVATED) {
        revert("Validator already activated");
    }
    require(validator.state == State.VALID, "KeysManager: Invalid Key");
    validator.state = State.ACTIVATED;
    emit ActivateValidator(publicKeys);
}
```

Figure 4.1: `contracts/KeysManager.sol#L80-L88`

Recommendations

Short term, modify the `ActivateValidator` event such that it specifies only the relevant public key, or introduce an additional `index` parameter to signify which of the public keys in the array was activated.

Long term, ensure that events are emitted for all critical system operations and that those events include relevant information and could serve as an audit log for operators in the event of an attack or system failure.

5. Lack of events for critical operations	
Severity: **Informational**	Difficulty: **Medium**
Type: Auditing and Logging	Finding ID: TOB-PER2-5
Targets: `contracts/Core.sol`, `contracts/CoreRef.sol`, `contracts/Issuer.sol`, `contracts/Oracle.sol`	

Description

Various functions in the `Core`, `CoreRef`, `Issuer`, and `Oracle` contracts do not emit events for critical system operations. Without events, it can be difficult to monitor the behavior of the contracts; if one of the contracts experienced an attack or system failure, for example, there would not be a sufficient audit log, and its operators would need to perform an error-prone analysis of the transactions sent to the contract to address the issue.

```
function set(bytes32 _key, address _address) external override onlyGovernor {
    coreContract[_key] = _address;
}
```

Figure 5.1: `contracts/Core.sol#L73-L75`

The following functions should emit events:

- `Core.setWithdrawalCredential`
- `Core.set`
- `CoreRef.setCore`
- `Issuer.setMinActivatingDeposit`
- `Issuer.setPendingValidatorsLimit`
- `Issuer.updatePendingValidator`
- `Oracle.updateBeaconChainData`
- `Oracle.updateValidatorQuorom`
- `Oracle.updateValidatorActivationDuration`

Recommendations

Short term, add events for all critical operations that result in state changes. Events aid in contract monitoring and the detection of suspicious behavior.

Long term, ensure that the events emitted for critical system operations include relevant information and could serve as an audit log for operators in the event of an attack or system failure. Use Slither to identify functions that do not emit events for critical operations.

6. Inconsistent VALIDATOR_DEPOSIT values

Severity: **Undetermined**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-PER2-6
Targets: `contracts/Issuer.sol`, `contracts/KeysManager.sol`	

Description

The `Issuer` and `KeysManager` contracts define a public constant, `VALIDATOR_DEPOSIT`, indicating the amount of ETH that each validator is expected to deposit into the system. However, the two contracts define different values for this constant.

```
uint256 public constant VALIDATOR_DEPOSIT = 31e18;
```

Figure 6.1: `contracts/Issuer.sol#L14`

```
uint256 public constant VALIDATOR_DEPOSIT = 32e18;
```

Figure 6.2: `contracts/KeysManager.sol#L16`

We were unable to determine the impact of this issue, but it could lead to undefined behavior.

Recommendations

Short term, address the inconsistent definitions of the `VALIDATOR_DEPOSIT` constant in the `Issuer` and `KeysManager` contracts.

7. StakingPool.slash function is vulnerable to sandwich attacks

Severity: **High**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-PER2-7
Targets: `contracts/StakingPool.sol`	

Description

When a user's stake is slashed, the `Oracle` contract calls the `StakingPool` contract's `slash` function, which calls `Router.swapExactTokensForTokens` to swap `pstake` for `StkEth`. The `amountOutMin` parameter of `swapExactTokensForTokens` can be set to zero, so the `Oracle` may not receive any `StkEth` in the trade. By sandwiching such a trade, a malicious bot could cause the `Oracle` contract to experience a high amount of slippage and to incur a loss of funds.

```
    function slash(uint256 amount) external override {
    [...]
        if(amountsIn[0] > pstakeBalance){
            pstake.approve(address(router), pstakeBalance);
            router.swapExactTokensForTokens(pstakeBalance, 0, path, address(this),
block.timestamp + 100);
        }
     [...]
    }
```

Figure 7.1: `contracts/StakingPool.sol#L96-L123`

Specifically, a malicious bot could keep track of the `Oracle`'s `slash` transactions and front-run and back-run one of the `Oracle`'s transactions with its own buy and sell transactions. This would temporarily inflate the price of the asset and lead to a loss of funds.

Exploit Scenario

A malicious bot front-runs and back-runs a transaction sent by the `Oracle` with buy and sell transactions. By sandwiching the `Oracle`'s trade, the bot causes it to incur a loss.

Recommendations

Short term, have the `Oracle` contract set the value of `amountOutMin` to prevent sandwich attacks and significant slippage.

8. Erroneous state variable shadowing

Severity: **Undetermined**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-PER2-8
Targets: `contracts/Issuer.sol`	

Description

The `Issuer` contract's `activatingDeposit` and `pendingValidatorLimit` functions are meant to return state variables (`minActivatingDeposit` and `pendingValidatorsLimit`, respectively). However, the contract defines uninitialized local variables of the same names, which shadow those state variables. As a result, the functions return the uninitialized values instead of the underlying state variables. Moreover, `minActivatingDeposit` and `pendingValidatorsLimit` are declared public, so they have default getter implementations to fetch the state variables.

```
    function activatingDeposit()
        public
        view
        returns (uint256 minActivatingDeposit)
    {
        return minActivatingDeposit;
    }

    /// @notice function returns pending validator limit.
    /// @return pendingValidatorsLimit number of pending validators.
    function pendingValidatorLimit()
        public
        view
        returns (uint256 pendingValidatorsLimit)
    {
        return pendingValidatorsLimit;
    }
```

Figure 8.1: `contracts/Issuer.sol#L51-L68`

We were unable to determine the impact of this issue. However, it could lead to undefined behavior, especially if the values are used by off-chain components.

Recommendations

Short term, if the `activatingDeposit` and `pendingValidatorLimit` functions simply return the `minActivatingDeposit` and `pendingValidatorsLimit` variables without first operating on them, use the default getter functions to fetch those variables, and remove the `activatingDeposit` and `pendingValidatorLimit` functions. This will save gas during the contract's deployment.

Long term, use Slither to catch instances of unintended variable shadowing.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system’s resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix lists code quality findings that we identified through a manual review.

- **Remove the redundant AddressSet check.** The Oracle contract's removeOracleMember method is called to remove an oracle member and checks whether the AddressSet contains the address of that oracle member before removing it. However, the remove method of the AddressSet returns a boolean indicating whether an item is included in the set or has already been removed. Thus, the check performed by the Oracle can be removed.

```
function removeOracleMember(address oracleMemberToDelete)
    external
    override
    onlyGovernor
{
    require(oracleMembers.contains(oracleMemberToDelete), "Oracle member not
present");
    oracleMembers.remove(oracleMemberToDelete);
    emit oracleMemberRemoved(oracleMemberToDelete, oracleMemberLength());
}
```

Figure C.1: contracts/Oracle.sol#L260-L268

- **Refactor the magic numbers into a defined constant.** The pendingValidatorsLimit, which must be set to at least 10,000, is defined in multiple parts of the codebase and should likely be refactored into a constant defined in a single location.

```
constructor(
    address core,
    uint256 _minActivatingDeposit,
    uint256 _pendingValidatorsLimit,
    address demoDeposit
) CoreRef(core) {
    DEPOSIT_CONTRACT = IDepositContract(demoDeposit);
    minActivatingDeposit = _minActivatingDeposit;

    require(_pendingValidatorsLimit < 10000, "Issuer: invalid limit");
```

Figure C.2: contracts/Issuer.sol#L34-L43

```
function setPendingValidatorsLimit(uint256 _pendingValidatorsLimit)
```

```
    external
    onlyGovernor
{
    require(_pendingValidatorsLimit < 10000, "Issuer: invalid limit");
```

Figure C.3: `contracts/Issuer.sol#L80-L84`

- **Consider making `Oracle.DEPOSIT_LIMIT` a constant to save gas**.

```
uint256 public DEPOSIT_LIMIT = 32e18;
```

Figure C.4: `contracts/Oracle.sol#L20`

- **Consider replacing certain `revert` statements with `require` statements.** Throughout the codebase, `revert` statements are used with `if/else` conditional statements (figure C.5). These `revert` statements should be replaced by `require` statements.

```
if (validator.state == State.ACTIVATED) {
    revert("Validator already activated");
}
```

Figure C.5: `contracts/KeysManager.sol#L82-L84`

- **The `ifMinterSelf` modifier in `CoreRef.sol` is a no-op.** The `ifMinterSelf` modifier is not being used and should be removed from the codebase.

```
modifier ifMinterSelf() {
    if (_core.isMinter(address(this))) {
        _;
    }
}
```

Figure C.6: `contracts/CoreRef.sol#L18-L22`

- **Remove the redundant conditional check in the `KeysManager` contract's `activateValidator` function.**

```
if (validator.state == State.ACTIVATED) {
    revert("Validator already activated");
}
require(validator.state == State.VALID, "KeysManager: Invalid Key");
```

Figure C.7: `contracts/KeysManager.sol#L82-L85`

- **In the `Oracle` contract, use a `require` statement instead of using both an `if` statement and a `revert` statement in the same line of code.** In two parts of the codebase, the `Oracle` contract checks a condition and calls the `revert` function if the condition evaluates to `false`. These `if` and `revert` statements can be replaced by a single `require` statement.

```
if (isOracle(msg.sender) == false) revert("Not oracle Member");
```

Figure C.8: `contracts/Oracle.sol#L340`

```
if (isOracle(msg.sender) == false) revert("Not oracle Member");
```

Figure C.9: `contracts/Oracle.sol#L387`

- **Address the incorrect revert message returned by the `KeysManager` contract.** When activating a validator, the `KeysManager` contract checks that the `msg.sender` is the `Oracle`. However, the `require` statement for that condition returns a misleading message if the condition is not met; the message indicates the `msg.sender` should be an issuer.

```
require(
    msg.sender == core().oracle(),
    "KeysManager: Only issuer can activate"
);
```

Figure C.10: `contracts/KeysManager.sol#L76-L79`

- **Remove all unused and dead code from the codebase.** See figures C.11–C.18 for examples of commented-out code and an unused variable, function, and modifier.

```
// require(_isEmptySigningKey(publicKey), "KeysManager: empty signing key");
// _validator = validator;
```

Figure C.11: `contracts/KeysManager.sol#L58-L59`

```
/// event MintStkEthForEth (uint256 amount, address user, uint256 stkEthToMint);
```

Figure C.12: `contracts/Issuer.sol#L24`

```
/// emit MintStkEthForEth (amount, user,stkEthToMint);
```

Figure C.13: `contracts/Issuer.sol#L112`

```
// function stake(uint256 amount) external {

//     pstake.transferFrom(_msgSender(), address(this), amount);

//     uint256 shares = calcShares(amount);
//     userShare[_msgSender()] = userShare[_msgSender()] + shares;

//     totalShares = totalShares + shares;
//     totalStake = totalStake + amount;
// }


// function calcShares(uint256 amount) public view returns (uint256 shares){
//     if(totalStake == 0 && totalShares == 0){
//         shares = amount;
//     }else{
//         shares = amount * totalShares/totalStake;
//     }
// }
```

Figure C.14: `contracts/StakingPool.sol#L74-L92`

```
/// @notice function for checking if signing key ...
/// @param _key ...
function _isEmptySigningKey(bytes memory _key)
    internal
    pure
    returns (bool)
{
[...]
}
```

Figure C.15: `contracts/KeysManager.sol#L197-L215`

```
uint64 lastCompletedTimeFrame;
```

Figure C.16: `contracts/Oracle.sol#L32`

```
modifier ifMinterSelf() {
    if (_core.isMinter(address(this))) {
        _;
    }
}
```

Figure C.17: `contracts/CoreRef.sol#L18-L22`

```
    // Appointed as a governor so guardian can have indirect access to revoke ability
    _setupGovernor(address(this));
```

Figure C.18: `contracts/Permissions.sol#L16-L17`

- **Pack the `Oracle` contract's state variables to save gas.**

```
modifier ifMinterSelf() {
    if (_core.isMinter(address(this))) {
        _;
    }
}
```

Figure C.19: `contracts/Oracle.sol#L32-L42`

- **The `IStkEthIssuer` interface is not used in the codebase and should be removed.**

```
pragma solidity ^0.8.0;

import "./ICoreRef.sol";

/// @title StkEthIssuer interface
/// @author Ankit Parashar
interface IStkEthIssuer is ICoreRef{

    function stake() payable external returns(uint256 amount);

}
```

Figure C.20: `contracts/IStkEthIssuer.sol#L1-L12`

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General Security Considerations

- ❑ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❑ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❑ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❑ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.

- ❑ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❑ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.

- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

E. Fix Log

On March 14, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Persistence team for the issues identified in this report. The fixes were merged into the main branch of the project repository, and we worked from commit hash `8f4f6fc5166aff166173229fde42ddfe6a0c8fdb`.

We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, see the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Missing checks of common ERC20 functions' return values	Medium	Partially Fixed
2	Lack of checks for the zero address	Low	Partially Fixed
3	Core's init method is prone to front-running	High	Fixed
4	Emission of duplicate events in a loop	Informational	Fixed
5	Lack of events for critical operations	Informational	Partially Fixed
6	Inconsistent VALIDATOR_DEPOSIT values	Undetermined	Fixed
7	StakingPool.slash function is vulnerable to sandwich attacks	High	Fixed
8	Erroneous state variable shadowing	Undetermined	Fixed

Detailed Fix Log

TOB-PER2-1: Missing checks of common ERC20 functions' return values
Partially Fixed. The `Oracle` contract now checks the results of calls to the `approve` function, and the `StakingPool` contract now checks the results of calls to the `transferFrom` function. However, the `StakingPool` contract does not check the values returned by the `approve` and `transfer` functions of the `pstake` and `StkEth` contracts. Implementing checks of the values returned by these first-party tokens would sufficiently address the issue described in this finding; however, the Persistence team should consider using OpenZeppelin's `SafeERC20` transfer functions for interactions with other nonstandard ERC20 tokens.

TOB-PER2-2: Lack of checks for the zero address
Partially Fixed. The Persistence team added zero-address checks for the `_weth` parameter of the `StakingPool` contract's `initialize` function. However, other arguments of this function, and of other functions mentioned in the finding (e.g., the `Core` contract's `set` function), are still not checked against the zero address.

TOB-PER2-3: Core's init method is prone to front-running
Fixed. The constructor of the `Core` contract now calls the `init` function, which prevents front-running attacks.

TOB-PER2-4: Emission of duplicate events in a loop
Fixed. The `activateValidator` function no longer emits duplicate events.

TOB-PER2-5: Lack of events for critical operations
Partially Fixed. Most of the functions mentioned in this finding (all but the `Oracle` functions) now emit events.

TOB-PER2-6: Inconsistent VALIDATOR_DEPOSIT values
Fixed. The Persistence team addressed the inconsistent definitions of `VALIDATOR_DEPOSIT` by updating the definition in the `KeysManager` contract.

TOB-PER2-7: StakingPool.slash function is vulnerable to sandwich attacks
Fixed. The `slash` method will not execute a swap if the purchase price deviates from the target price by more than 5%.

TOB-PER2-8: Erroneous state variable shadowing
Fixed. The Persistence team removed the names of the local return variables. This change will prevent collisions between those variable names and the identically named state variables.