# SMART CONTRACT AUDIT REPORT

for

# ELEMENT FINANCE

Prepared By: Shuxiao Wang

PeckShield

April 14, 2021

## Document Properties

| Client | Element Finance |
|---|---|
| Title | Smart Contract Audit Report |
| Target | Element Finance |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Yiqun Chen, Jeff Liu, Xuxian Jiang |
| Reviewed by | Jeff Liu |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | April 14, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc | April 12, 2021 | Xuxian Jiang | Release Candidate |
| 0.2 | April 6, 2021 | Xuxian Jiang | Additional Findings |
| 0.1 | April 1, 2021 | Xuxian Jiang | Initial Draft |

## Contact

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

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

# Contents

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **Element Finance** protocol, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

## 1.1 About Element Finance

`Element Finance` aims to bring high fixed yield rates to the DeFi market with a focus on `BTC`, `ETH`, and `USDC`. The fixed yield is collateralized by variable yield positions and the higher rates offered as fixed yield are driven and maintained by various market forces. Specifically, `Element` takes a novel approach to fixed yield, splitting the principal and interest of existing yield positions into separate, fungible tokens which are designed to be traded or staked in various AMMs. It also improves capital efficiency by allowing users to sell their principal as a fixed yield position, further leveraging or increasing exposure to interest without liquidation risk. In other words, users can gain more by simply staking their unused LP tokens or principal and then realizing trading fees as additional revenue.

The basic information of Element Finance is as follows:

Table 1.1: Basic Information of Element Finance

| Item | Description |
|---:|---|
| Issuer | Element Finance |
| Website | https://element.fi/ |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | April 14, 2021 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories (e.g., `balancer-core-v2`) and this audit relies on the correctness and safety of the associated `balancer-core-v2` protocol, which is not part of this audit.

- https://github.com/element-fi/elf-contracts.git (eed3695)

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

- https://github.com/element-fi/elf-contracts.git (39194a5)

## 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 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.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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.

## 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

Table 1.3: The Full List of Check Items

| Category | Check Item |
|---|---|
| Basic Coding Bugs | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead Of Transfer |
| | Costly Loop |
| | (Unsafe) Use Of Untrusted Libraries |
| | (Unsafe) Use Of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| Semantic Consistency Checks | Semantic Consistency Checks |
| Advanced DeFi Scrutiny | Business Logics Review |
| | Functionality Checks |
| | Authentication Management |
| | Access Control & Authorization |
| | Oracle Security |
| | Digital Asset Escrow |
| | Kill-Switch Mechanism |
| | Operation Trails & Event Generation |
| | ERC20 Idiosyncrasies Handling |
| | Frontend-Contract Integration |
| | Deployment Consistency |
| | Holistic Risk Management |
| Additional Recommendations | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

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. |

bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

# 2 Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the Element 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 | 1 | |
| Low | 2 | |
| Informational | 2 | |
| Total | 5 | |

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

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Element Finance Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Medium | safeTransfer()/safeApprove() Replacement | Coding Practices | Resolved |
| PVE-002 | Low | Non-Compliant ERC20 Implementation Of Tranche And InterestToken | Business Logic | Fixed |
| PVE-003 | Low | Improved mint() Logic in UserProxy | Business Logic | Fixed |
| PVE-004 | Informational | Suggested Addition of recoverERC20() in UserProxy | Business Logic | Resolved |
| PVE-005 | Informational | Suggested Use of Differentiating Event Names | Error Conditions, Return Values, Status Codes | Resolved |

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 safeTransfer()/safeTransferFrom()/safeApprove() Replacement

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

### 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 3.1: USDT::transferFrom()

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 `approve()/transfer()` as well, i.e., `safeApprove()/safeTransfer()`.

In current implementation, if we examine the `WrappedPosition::deposit()` routine that is designed to deposit tokens into the Wrapped Position contract from the participating user. To accommodate the specific idiosyncrasy, there is a need to user `safeTransferFrom()`, instead of `transferFrom()` (line 86).

```
74     /// @notice Entry point to deposit tokens into the Wrapped Position contract
75     ///         Transfers tokens on behalf of caller so the caller must set
76     ///         allowance on the contract prior to call.
77     /// @param _amount The amount of underlying tokens to deposit
78     /// @param _destination The address to mint to
79     /// @return Returns the number of Wrapped Position tokens minted
80     function deposit(address _destination, uint256 _amount)
81         external
82         override
83         returns (uint256)
84     {
85         // Send tokens to the proxy
86         token.transferFrom(msg.sender, address(this), _amount);
87         // Calls our internal deposit function
88         (uint256 shares, ) = _deposit();
89         // Mint them internal ERC20 tokens corresponding to the deposit
90         _mint(_destination, shares);
91         return shares;
92     }
```

Listing 3.2: WrappedPosition::deposit()

In the meantime, we also suggest to use the safe-version of `transfer()/transferFrom()` in other related routines, including `Tranche::deposit()`, `UserProxy::mint()`, `YVaultAssetProxy::reserveDeposit`

`()`, and `reserveWithdraw()`.

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

**Status** The team has confirmed that the `Element` protocol is designed not to support ERC20 tokens that are non-standard and do not revert on transfer from.

## 3.2 Non-Compliant ERC20 Implementation Of Tranche And InterestToken

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The `Element` protocol takes a novel approach to meet the fixed yield goal by splitting the principal and interest of existing yield positions into separate, fungible tokens. These fungible tokens can then be traded or staked in various AMMs. Naturally, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of both principal and interest tokens.

Our analysis shows that the current implementation does not strictly follow the ERC20 specification and may cause unnecessary incompatibility issue. Both principal and interest tokens are inherited from the `ERC20Permit` contract, which somehow does not define or maintain the `totalSupply` of respective tokens. To elaborate, we show below the code snippet from `ERC20Permit`.

```
abstract contract ERC20Permit is IERC20Permit {
    // --- ERC20 Data ---
    string public name;
    string public override symbol;
    uint8 public override decimals;

    mapping(address => uint256) public override balanceOf;
    mapping(address => mapping(address => uint256)) public override allowance;
    mapping(address => uint256) public override nonces;

    // --- EIP712 niceties ---
    // solhint-disable-next-line var-name-mixedcase
    bytes32 public override DOMAIN_SEPARATOR;
    // bytes32 public constant PERMIT_TYPEHASH = keccak256("Permit(address owner,address
        spender,uint256 value,uint256 nonce,uint256 deadline)");
```

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

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

```
    bytes32
        public constant PERMIT_TYPEHASH = 0
            x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a169c64845d6126c9;

    constructor(string memory name_, string memory symbol_) {
        name = name_;
        symbol = symbol_;
        decimals = 18;

        balanceOf[address(0)] = type(uint256).max;
        balanceOf[address(this)] = type(uint256).max;

        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256(
                    "EIP712Domain(string name,string version,uint256 chainId,address
                        verifyingContract)"
                ),
                keccak256(bytes(name)),
                keccak256(bytes("1")),
                _getChainId(),
                address(this)
            )
        );
```

```
207     }
```

Listing 3.3: USDT::transferFrom()

Moreover, it comes to our attention that the `constructor()` routine has two special addresses initialized with the `type(uint256).max` balance (lines 193 – 194). It is suggested to better clarify the purpose of these two addresses as they cause unnecessary inconsistency, i.e., the balance sum of all possible accounts is not the same as the commonly-conceived `totalSupply`.

**Recommendation** Be consistent with the widely adopted ERC20 specification in both principal and interest token contracts.

**Status** This issue has been fixed in this commit: `3c93d9a`.

## 3.3 Improved mint() Logic in UserProxy

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

### Description

To facilitate user on-boarding, the Element Finance protocol has a convenience `UserProxy` to consolidate actions needed to create interest or principal tokens to one call and further hold user allowances for asset transfers. While examining the `UserProxy` contract, we notice a possible improvement on current implementation.

To elaborate, we show below its `mint()` function. This function is designed to mint a principal and interest token pair from either underlying token or `ETH` and then return the tokens to the caller. It comes to our attention that if the underlying token is not `ETH` (in the `else`-branch at lines 136 – 141), we need to add the following requirement to ensure that the calling user will not accidentally send `ETH`: `require(msg.value ==0, "Incorrect amount provided")`. Note the accidentally sent `ETH` may be locked in the `UserProxy` contract. The only way to uncover is to perform the `selfdestruct` operation via `deprecate()`, which seems an overkill.

```
119     function mint(
120         uint256 _amount,
121         IERC20 _underlying,
122         uint256 _expiration,
123         address _position,
124         PermitData[] calldata _permitCallData
125     ) external payable notFrozen() preApproval(_permitCallData) {
```

```
        // If the underlying token matches this predefined 'ETH token'
        // then we create weth for the user and go from there
        if (address(_underlying) == _ETH_CONSTANT) {
            // Check that the amount matches the amount provided
            require(msg.value == _amount, "Incorrect amount provided");
            // Create weth from the provided eth
            weth.deposit{ value: msg.value }();
            weth.transfer(address(_position), _amount);
            // Proceed to internal minting steps
            _mint(_expiration, _position);
        } else {
            // Move the user's funds to the wrapped position contract
            _underlying.transferFrom(msg.sender, address(_position), _amount);
            // Proceed to internal minting steps
            _mint(_expiration, _position);
        }
    }
```

Listing 3.4: UserProxy::mint()

**Recommendation** Improve the `mint()` logic to prevent accidentally sent assets from being locked in the `UserProxy` contract.

**Status** This issue has been fixed in this commit: `3c93d9a`.

## 3.4 Suggested Addition of recoverERC20() in UserProxy



- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UserProxy`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

By design, the Element Finance protocol has developed a number of contracts that hold various types of assets. From past experience with current popular DeFi protocols, e.g., `YFI/Curve`, we notice that there is always non-trivial possibilities that non-related tokens may be accidentally sent to the pool contract(s). To avoid unnecessary loss of `Element` users, we suggest to add necessary support of rescuing tokens accidentally sent to the contract. This is a design choice for the benefit of protocol users.

**Recommendation** Add the support of rescuing tokens accidentally sent to the contract. An example addition is shown below:

```
function recoverERC20(address _token, uint256 _amount) external onlyOwner {
    IERC20(_token).safeTransfer(owner(), _amount);
    emit Recovered(_token, _amount);
}
```

Listing 3.5: UserProxy::recoverERC20()

**Status** The team is aware of this issue and considers the expectation that the governance will rescue funds "both exposes it to liability and is un-scalable."

## 3.5 Suggested Use of Differentiating Event Names

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ConvergentPoolFactory`
- Category: Status Codes [6]
- CWE subcategory: CWE-391 [2]

### Description

In `Ethereum`, 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 show the `create()` routine from the `ConvergentPoolFactory` contract. This routine is designed to deploy a new `ConvergentPool` instance and then register the new instance. It comes to our attention that the `_register()` call (line 63) in essence emits an event of `PoolRegistered (pool)` where the `pool` is the new deployed instance address. Interestingly, the `constructor` of `ConvergentCurvePool` automatically registers the new pool via the `vault.registerPool(IVault.PoolSpecialization .TWO_TOKEN)`, which emits another event in the following form: `emit PoolRegistered(poolId)`, where `poolId` denotes the assigned pool ID in `byte32`. These two events come with the same name, but with different parameters. To avoid unnecessary confusion, it is suggested to use self-differentiating event names.

```
40      function create(
41          address _underlying,
42          address _bond,
43          uint256 _expiration,
44          uint256 _unitSeconds,
45          uint256 _percentFee,
46          string memory _name,
```

```
        string memory _symbol
    ) external returns (address) {
        address pool = address(
            new ConvergentCurvePool(
                IERC20(_underlying),
                IERC20(_bond),
                _expiration,
                _unitSeconds,
                vault,
                _percentFee,
                percentFeeGov,
                governance,
                _name,
                _symbol
            )
        );
        _register(pool);
        return pool;
    }
```

Listing 3.6: ConvergentPoolFactory::create()

```
    function registerPool(PoolSpecialization specialization)
        external
        override
        nonReentrant
        noEmergencyPeriod
        returns (bytes32)
    {
        // Use _totalPools as the Pool ID nonce. uint80 assumes there will never be more
            than than 2**80 Pools.
        bytes32 poolId = _toPoolId(msg.sender, specialization, uint80(_poolNonce.current
            ()));
        require(!_isPoolRegistered[poolId], "INVALID_POOL_ID"); // Should never happen

        _poolNonce.increment();
        _isPoolRegistered[poolId] = true;

        emit PoolRegistered(poolId);
        return poolId;
    }
```

Listing 3.7: PoolRegistry::registerPool()

**Recommendation** Revise the above events by properly choosing different names as they encode different information.

**Status** Since these events are emitted by different contract addresses in different contexts, the team considers that they are still distinguishable and plans to leave it as is.

# 4 Conclusion

In this audit, we have analyzed the `Element Finance` design and implementation. The system presents a unique offering in bringing high fixed yield rates to the DeFi market. Specifically, it enables fixed yield by splitting the principal and interest of existing yield positions into separate, fungible tokens which are designed to be traded or staked in various AMMs. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

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

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

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

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.