# SMART CONTRACT AUDIT REPORT

for

# AUDITCHAIN



Prepared By: Shuxiao Wang

Hangzhou, China
November 26, 2020

## Document Properties

| | |
|---|---|
| Client | Auditchain |
| Title | Smart Contract Audit Report |
| Target | Auditchain |
| Version | 1.0-rc |
| Author | Xuxian Jiang |
| Auditors | Huaguo Shi, Jeff Liu, Xuxian Jiang |
| Reviewed by | Jeff Liu |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0-rc | November 26, 2020 | Xuxian Jiang | Release Candidate |
| 0.2 | November 22, 2020 | Xuxian Jiang | Additional Findings |
| 0.1 | November 20, 2020 | 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 **Auditchain**, 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 Auditchain

`Auditchain` is leading the development of the `DCARPE Assurance and Disclosure Protocol`, (the "Protocol"). The Protocol proposes to enable real time or near real time financial reporting as well as the continuous audit of (i) compliance with functional objectives of enterprise systems and organizational controls, (ii) data structure and accuracy and (iii) disclosure control architecture and compliance objectives with `Generally Accepted Accounting Principles` in the USA ("GAAP") and `International Financial Reporting Standards` outside the USA, ("IFRS"). Auditchain allows for enterprises to provide stakeholders with the highest levels of assurance through decentralized consensus-based enterprise external validation.

The basic information of Auditchain is as follows:

Table 1.1: Basic Information of Auditchain

| Item | Description |
|---|---|
| Issuer | Auditchain |
| Website | https://www.auditchain.com |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | November 26, 2020 |

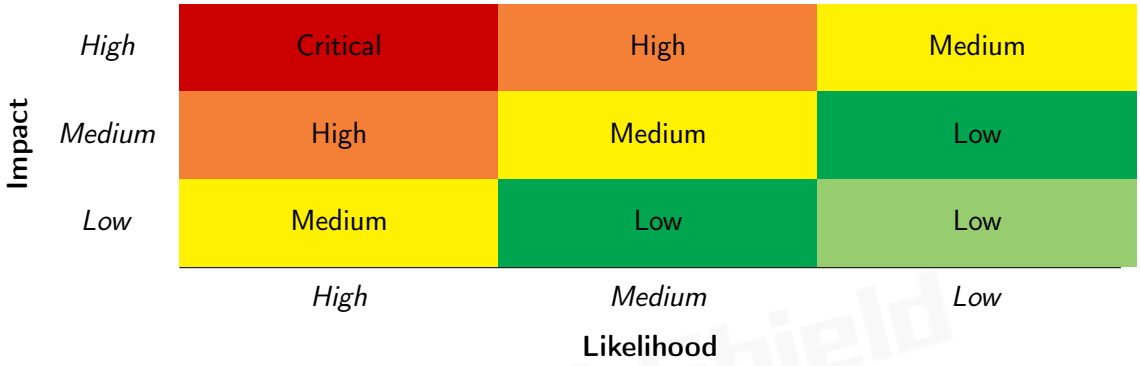
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/DAOCapsule/AUDT-Capsule-Lift-Off.git (c199da7)

## 1.2 About PeckShield

PeckShield Inc. [16] 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




## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

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

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

Table 1.3: The Full List of Check Items

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

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

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business Logic** | Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. |
| **Initialization and Cleanup** | Weaknesses in this category occur in behaviors that are used for initialization and breakdown. |
| **Arguments and Parameters** | Weaknesses in this category are related to improper use of arguments or parameters within function calls. |
| **Expression Issues** | Weaknesses in this category are related to incorrectly written expressions within code. |
| **Coding Practices** | Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained. |

# 2 Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the Auditchain implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 0 | |
| Medium | 1 | ■ |
| Low | 3 | ■■■ |
| Informational | 1 | ■ |
| Total | 5 | |

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Auditchain Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Oversized totalReward May Lock User Stakes | Numeric Errors | Fixed |
| PVE-002 | Low | Improved Sanity Checks For System Parameters | Coding Practices | Fixed |
| PVE-003 | Informational | AUDT Tokens Pausable For Migration, But Not Transfer | Business Logic | Confirmed |
| PVE-004 | Medium | Burnability of Locked Accounts | Business Logic | Confirmed |
| PVE-005 | Low | Suggested Uses of SafeMath | Coding Practices | Fixed |

Beside the identified issues, upon the observation that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.6` instead of `pragma solidity ^0.6.6`.

In the meantime, 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 Oversized totalReward May Lock User Stakes

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Staking`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [3]

### Description

The Auditchain protocol allows `AUDT` holders stake their tokens for rewards. Specifically, users can stake `AUDT`s into the pool and get tradable ERC20-compliant staking receipts. With staking receipts, the user can redeem for a proportional share of the pool (with pre-configured `totalReward`) and a ratio of the governance token, i.e., `DCAP`, pursuant to the agreed minting schedule after the expiration of the staking period. If redeemed before the expiration, the user will get their staked amount back.

To elaborate, we show below the `redeem()` logic. If we examine the redemption logic after the staking expiration, the users can expect to receive additional rewards beyond the previously staked amounts (line 252).

```
240     /**
241     * @dev Function to redeem contribution. Based on the staking period function may
            send rewards or just deposit.
242     * If user redeems after staking ended, reward will be added to deposit. If staking
            is still in progress,
243     * user only receives amount contributed.
244     * @param amount number of tokens being redeemed
245     */
246    function redeem(uint256 amount) public {
247
248        require(_stakingToken.balanceOf(msg.sender) >= amount, "Staking:redeem - you are
               claiming more than your balance.");
249        _burnStakedToken(amount);
250
251        if (block.number > stakingDateEnd){
```

```
252             _deliverRewards(amount);
253             emit LogTokensRedeemed(msg.sender, returnEarningsPerAmount(amount));
254         }
255         else{
256             _returnDeposit(amount);
257             emit LogTokensRedeemed(msg.sender, amount);
258         }
259     }
```

Listing 3.1: Staking :: redeem()

The rewarding logic is implemented in `_deliverRewards()`. In this helper routine, it firstly computes the `amountRedeemed` (line 279) that will be returned back to the user. This amount is computed as `amount * returnEarningRatio()).div(1e18)` in `returnEarningsPerAmount()`.

```
271     /**
272      * @dev Function to deliver rewards called from redeem() function
273      * @param amount number of tokens to deliver (token originally deposited + staking
             rewards)
274      */
275     function _deliverRewards(uint256 amount) internal {
276
277         uint256 amountRedeemed;
278
279         amountRedeemed = returnEarningsPerAmount(amount);
280         released[msg.sender] = released[msg.sender].add(amountRedeemed);
281         totalReleased = totalReleased.add(amountRedeemed);
282         _auditToken.safeTransfer(msg.sender, amountRedeemed);
283         _deliverGovernanceToken((governanceTokenRatio * amount) / 1e18);
284         LogRewardDelivered(msg.sender, amountRedeemed);
285     }
```

Listing 3.2: Staking :: _deliverRewards()

Next, if we follow the execution logic, `returnEarningRatio()` returns `(totalReward.mul(1e18)/stakedAmount)+ 1e18` (line 154) after staking expiration. We notice the multiplication of `totalReward.mul(1e18)` may overflow if `totalReward` is initially configured unreasonably large. The overflow consequence directly reverts every redemption attempt if the staking period ends.

```
145     /**
146      * @dev Function to return earning ratio
147      * @return number representing earning ratio with precision to 18 decimal values
148      */
149     function returnEarningRatio() public view returns (uint256) {
150
151         if (stakedAmount == 0)
152             return totalReward; // At this stage there is no contributions
153          else
154             return (totalReward.mul(1e18) / stakedAmount) + 1e18 ;
155     }
156
157      /**
```

```
158     * @dev Function to return earning ratio per given amount
159     * @param amount - amount in question
160     * @return number representing earning ratio for given amount
161     */
162    function returnEarningsPerAmount(uint256 amount) public view returns(uint256) {
163
164        return (amount * returnEarningRatio()).div(1e18);
165    }
```

Listing 3.3: Staking :: returnEarningsPerAmount()

**Recommendation** Validate the pre-configured `totalReward` to ensure no overflow may occur.

**Status** The issue has been fixed by this commit: `c69e52f`.

## 3.2 Improved Sanity Checks For System Parameters

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

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Auditchain protocol is no exception. Specifically, if we examine the `AlphaPerp` contract, it has defined the following parameters: `stakingDateStart`, `stakingDateEnd`, `totalReward`, and `governanceTokenRatio`. These parameters define the block height to start staking, the block height to stop staking, the total reward amount, as well as the governance token ratio for issuance, respectively.

Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `totalReward` will revert every `redeem()` operation after the staking period, hence locking user stakes.

To elaborate, we show below its code snippet of `updateStakingPeriods()`. This routine updates the block heights to start and stop staking. However, they can be improved to validate that the given `_stakingDateStart` and `_stakingDateEnd` fall in an appropriate range.

```
115     /**
116     * @dev Function to manually update staking periods
117     * @param _stakingDateStart - start date of staking
118     * @param _stakingDateEnd - end date of staking
```

```
119      */
120     function updateStakingPeriods(uint256 _stakingDateStart, uint256 _stakingDateEnd)
            public onlyOwner() {
121
122         require(_stakingDateEnd != 0, "Staking:constructor - Staking end date can't be 0
                " );
123         require(_stakingDateStart != 0, "Staking:constructor - Staking start date can't
                be 0" );
124         stakingDateStart = _stakingDateStart;
125         stakingDateEnd = _stakingDateEnd;
126
127     }
```

Listing 3.4: Staking::updateStakingPeriods()

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes. An example revision to `updateStakingPeriods()` is shown below.

```
115      /**
116      * @dev Function to manually update staking periods
117      * @param _stakingDateStart - start date of staking
118      * @param _stakingDateEnd - end date of staking
119      */
120     function updateStakingPeriods(uint256 _stakingDateStart, uint256 _stakingDateEnd)
            public onlyOwner() {
121
122         require(_stakingDateStart > block.number, "Staking:constructor - Staking start
                date is already passed" );
123         require(_stakingDateEnd > stakingDateStart, "Staking:constructor - Staking end
                date can't be smaller than stakingDateStart" );
124         stakingDateStart = _stakingDateStart;
125         stakingDateEnd = _stakingDateEnd;
126
127     }
```

Listing 3.5: Revised Staking::updateStakingPeriods()

**Status** The issue has been fixed by this commit: `c69e52f`.

## 3.3 AUDT Tokens Pausable For Migration, But Not Transfer

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GovernanceToken, Token`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [4]

### Description

Both `AUDIT` token and `DCAP` are ERC20-compliant tokens. Accordingly, there is a need for their contract implementations, i.e., `Token` and `GovernanceToken`, to follow the ERC20 specification. As part of our audit, 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. Since both token contracts share a similar implementation, we use `AUDIT` as the representative for the following discussion.

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 "Auditchain" | ✓ |
| **symbol()** | Is declared as a public view function | ✓ |
| | Returns the symbol by which the token contract should be known, for example "AUDT". 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 | ✓ |
| | Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment | ✓ |
| **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 | ✓ |

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited Auditchain. In the following two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted

ERC20 specification.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

| Item | Description | Status |
|---|---|---|
| **transfer()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token transfer status | ✓ |
| | Reverts if the caller does not have enough tokens to spend | ✓ |
| | Allows zero amount transfers | ✓ |
| | Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers) | ✓ |
| | Reverts while transferring to zero address | ✓ |
| **transferFrom()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token transfer status | ✓ |
| | Reverts if the spender does not have enough token allowances to spend | ✓ |
| | Updates the spender's token allowances when tokens are transferred successfully | ✓ |
| | Reverts if the from address does not have enough tokens to spend | ✓ |
| | Allows zero amount transfers | ✓ |
| | Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers) | ✓ |
| | Reverts while transferring from zero address | ✓ |
| | Reverts while transferring to zero address | ✓ |
| **approve()** | Is declared as a public function | ✓ |
| | Returns a boolean value which accurately reflects the token approval status | ✓ |
| | Emits Approval() event when tokens are approved successfully | ✓ |
| | Reverts while approving to zero address | ✓ |
| **Transfer()** event | Is emitted when tokens are transferred, including zero value transfers | ✓ |
| | Is emitted with the from address set to $address(0x0)$ when new tokens are generated | ✓ |
| **Approve()** event | Is emitted on any successful call to approve() | ✓ |

Meanwhile, we notice in the `transferFrom()` routine, there is a common practice that is missing but widely used in other ERC20 contracts. Specifically, when `msg.sender = _from`, the current `transferFrom()` implementation disallows the token transfer if `msg.sender` has not explicitly allows spending from herself yet. A common practice will whitelist this special case and allow `transferFrom()` if `msg.sender = _from` even there is no allowance specified.

```
639     /**
640      * @dev See {IERC20-transferFrom}.
641      *
642      * Emits an {Approval} event indicating the updated allowance. This is not
643      * required by the EIP. See the note at the beginning of {ERC20};
644      *
645      * Requirements:
646      * - `sender` and `recipient` cannot be the zero address.
```

```
647      * - `sender` must have a balance of at least `amount`.
648      * - the caller must have allowance for ``sender``'s tokens of at least
649      * `amount`.
650      */
651     function transferFrom(address sender, address recipient, uint256 amount) public
             virtual override returns (bool) {
652         _transfer(sender, recipient, amount);
653         _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "
                 ERC20: transfer amount exceeds allowance"));
654         return true;
655     }
```

Listing 3.6: flat /Token.sol

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

| Feature | Description | Opt-in |
|---|---|---|
| **Deflationary** | Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls | — |
| **Rebasing** | The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address | — |
| **Pausible** | The token contract allows the owner or privileged users to pause the token transfers and other operations | — |
| **Blacklistable** | The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited | ✓ |
| **Mintable** | The token contract allows the owner or privileged users to mint tokens to a specific address | ✓ |
| **Burnable** | The token contract allows the owner or privileged users to burn tokens of a specific address | ✓ |
| **Hookable** | The token contract allows the sender/recipient to be notified while sending/receiving tokens | — |
| **Permittable** | The token contract allows for unambiguous expression of an intended spender with the specified allowance in an off-chain manner (e.g., a permit() call to properly set up the allowance with a signature). | — |

We point out that both `AUDIT` token and `DCAP` are not pausable even though the contract `Pausable` is inherited. The `Pausable` feature is used for migration purpose only, not for the purpose of pausing the entire token.

**Recommendation** Improve the `transferFrom()` logic by considering the special case when `msg.sender` = `_from`. In the meantime, consider the support of `permit()` (in EIP-2612) for better integration and usability.

**Status** This issue has been confirmed.

## 3.4 Burnability of Locked Accounts

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GovernanceToken, Token`
- Category: Business Logic [7]
- CWE subcategory: CWE-754 [5]

### Description

As mentioned in Section 3.3, both `AUDIT` and `DCAP` are ERC20-compliant tokens. And the ERC20-compliance checks show that they are burnable, mintable, ownable, with the locking ability on a per-user basis.

To elaborate, we show below the code snippet of `ERC20Burnable`. Note that both `AUDIT` and `DCAP` token contracts directly inherit from `ERC20Burnable`. Although both `AUDIT` and `DCAP` support the locking of a particular user, there is no locking-related validation checks in `ERC20Burnable`. As a result, the locked account may still be able to burn their tokens.

```
819 abstract contract ERC20Burnable is Context, ERC20 {
820     /**
821      * @dev Destroys `amount` tokens from the caller.
822      *
823      * See {ERC20-_burn}.
824      */
825     function burn(uint256 amount) public virtual {
826         _burn(_msgSender(), amount);
827     }
828
829     /**
830      * @dev Destroys `amount` tokens from `account`, deducting from the caller's
831      * allowance.
832      *
833      * See {ERC20-_burn} and {ERC20-allowance}.
834      *
835      * Requirements:
836      *
837      * - the caller must have allowance for ``accounts``'s tokens of at least
838      * `amount`.
839      */
```

```
840     function burnFrom(address account, uint256 amount) public virtual {
841         uint256 decreasedAllowance = allowance(account, _msgSender()).sub(amount, "ERC20
                : burn amount exceeds allowance");
842
843         _approve(account, _msgSender(), decreasedAllowance);
844         _burn(account, amount);
845     }
846 }
```

Listing 3.7: flat /ERC20Burnable

**Recommendation** Validate whether the account is being locked when `burn()` or `burnFrom()` is called. The burn operation should not proceed if the account is being locked.

**Status** This issue has been confirmed.

## 3.5 Suggested Uses of SafeMath

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [3]

### Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. During our analysis of the staking logic in `Staking`, we notice several occasions whether `SafeMath` is not used. Examples include the arithmetic operations at lines 164, 202, 216, and 283.

In the following, we choose two examples. The first example is the computation in `returnEarningsPerAmount()`: `(amount * returnEarningRatio()).div(1e18)` (line 164). The multiplication of `amount * returnEarningRatio()` is not guarded for overflow. We should point out that this multiplication will not overflow in this particular usage scenario. However, it is always preferable to guarantee the overflow will always be detected and blocked.

```
159     /**
160     * @dev Function to return earning ratio per given amount
161     * @param amount - amount in question
162     * @return number representing earning ratio for given amount
163     */
164     function returnEarningsPerAmount(uint256 amount) public view returns(uint256) {
165
166         return (amount * returnEarningRatio()).div(1e18);
```

```
167     }
```

Listing 3.8: Staking :: returnEarningsPerAmount()

The second example is the computation in `stake()`: `stakedAmount += amount` (line 202). It is suggested to replace it with `stakedAmount = stakedAmount.add(amount)`.

```
202     function stake(uint256 amount) public {

204         require(amount >= 100e18, "Staking:stake - Minimum contribution amount is 100
                AUDT tokens");
205         require(stakingDateStart >= block.number, "Staking:stake - deposit period ended.
                ");
206         require(blacklistedAddress[msg.sender] == false, "This address has been
                blacklisted");
207         stakedAmount += amount;  // track tokens contributed so far
208         _receiveDeposit(amount);
209         _deliverStakingTokens( amount);
210         emit LogStakingTokensIssued(msg.sender, amount);
211     }
```

Listing 3.9: Staking :: stake()

**Recommendation** Make use of `SafeMath` in the above calculations to better mitigate possible overflows.

**Status** The issue has been fixed by this commit: `c69e52f`.

# 4 Conclusion

In this audit, we have analyzed the `Auditchain` design and implementation. The system presents a unique offering in enabling enterprises to provide stakeholders with the highest levels of assurance through decentralized consensus-based enterprise external validation. The current code base is well organized and those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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.

# 5 Appendix

## 5.1 Basic Coding Bugs

### 5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

### 5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

### 5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

### 5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [12, 13, 14, 15, 17].
- Result: Not found
- Severity: Critical

### 5.1.5 Reentrancy

- Description: Reentrancy [18] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

### 5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

### 5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

### 5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

### 5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

### 5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

### 5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

### 5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

### 5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

### 5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

### 5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

### 5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

### 5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

## 5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

## 5.3 Additional Recommendations

### 5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

### 5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

### 5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

### 5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

# References


[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

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

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[5] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

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

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

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

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

[12] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[13] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[14] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[15] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[17] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[18] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.