SMART CONTRACT AUDIT REPORT

for

PolkaCipher

Prepared By: Shuxiao Wang

PeckShield

May 21, 2021

Document Properties

Client	PolkaCipher
Title	Smart Contract Audit Report
Target	PolkaCipher
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	May 21, 2021	Yiqun Chen	Final Release
1.0-rc	April 29, 2021	Yiqun Chen	Release Candidate

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 source code of the PolkaCipher protocol and the associated token contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About PolkaCipher

PolkaCipher is a cross-chain privacy preserving project in the Polkadot ecosystem. The primary aim is to expand the applications of NFTs & DeFi and related use cases in business settings as well as to increase the reach of Web3 applications in blockchain economies via a seamless integration with PolkaCipher. To achieve that, there is a need to bridge the gap between an off-chain Internet economy and on-chain blockchain decentralized applications using PolkaCipher that relies on verified vendors to provide accurate and trusted data. By design, PolkaCipher, once ready, will be community owned and fully decentralized.

The basic information of PolkaCipher is as follows:

Table 1.1: Basic Information of PolkaCipher

Item	Description
Issuer	PolkaCipher
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	May 21, 2021

In the following, we show the Git repository and the commit hash value used in this audit:

- https://github.com/polkacipher/Contracts.git (c79c731)

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

- https://github.com/polkacipher/Contracts.git (b936867)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

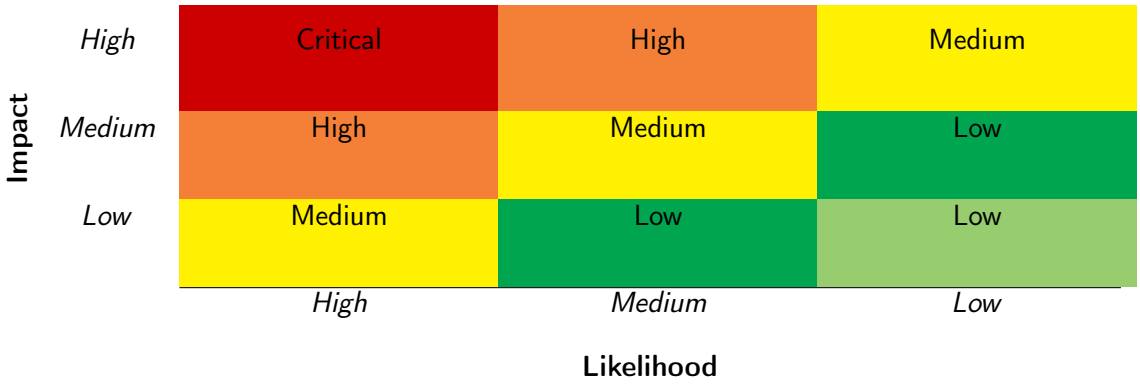
- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

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.

We perform the audit according to the following procedures:

- 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.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.2: Vulnerability Severity Classification



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 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.

1.4 Disclaimer

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

Table 1.3: 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
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the PolkaCipher. 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 ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	1	■
High	1	■
Medium	3	■ ■ ■
Low	2	■ ■
Informational	0	
Total	7	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. Note that the smart contract implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 critical-severity vulnerability, 1 high-severity vulnerability, 3 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key PolkaCipher Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Susceptible Replays Against MultisigWallet::execute()	Coding Practices	Fixed
PVE-002	High	Denial-Of-Service Against MultisigWallet::execute()	Coding Practices	Fixed
PVE-003	Medium	Reentrancy Risk In MultisigWallet::execute()	Coding Practices	Fixed
PVE-004	Critical	Funds Can Be Fully Drained From Vest	Security Features	Fixed
PVE-005	Low	safeTransfer()/safeTransferFrom() Replacement	Coding Practices	Fixed
PVE-006	Medium	Logic Error of getClaimable()	Business Logic	Fixed
PVE-007	Medium	Assumed Trust on Admin Keys	Security Features	Fixed

As a common suggestion, due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always preferred to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.8.0;` instead of `pragma solidity ^0.8.0;`.

Besides recommending specific countermeasures to mitigate these issues, we emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step 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 of the audited contract(s).

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	✓
	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 PolkaCipher. In the surrounding 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-

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	✓
Approval() event	Is emitted on any successful call to approve()	✓

adopted ERC20 specification. 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/ERC2222), 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	—
Pausable	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	—

4 Detailed Results

4.1 Susceptible Replays Against MultisigWallet::execute()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MultisigWallet`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

Description

The `PolkaCipher` protocol has a core `MultisigWallet` contract that allows approvers to collect their signatures for a given action. The `MultisigWallet` contract supports three approvers: Once the collected number of (verified) signatures is larger than the required threshold (saved in `_required`), the intended action will be executed.

```
37      function execute(address to, uint256 value, bytes memory data, bytes memory sig1,
            bytes memory sig2, bytes memory sig3) public payable returns (bool success) {
38          bytes32 hash = keccak256(abi.encodePacked(nonce, to, value ,data));
39          uint8 approvals;
40          approvals += _voters[hash.recover(sig1)];
41          approvals += _voters[hash.recover(sig2)];
42          approvals += _voters[hash.recover(sig3)];
43          if (approvals >= _required){
44              (success,) = to.call{value: value}(data);
45          }
46          nonce++;
47      }
```

Listing 4.1: MultisigWallet :: execute()

In particular, we show above the `execute()` function. The function generates the `hash` using required fields, i.e., `nonce`, `to`, `value`, and `data`. However, this calculation assumes all signatures produced by different signers are in the same chain. The absence of a `EIP-712 domainSeparator` with the `EIP-155 chainID` in current calculation makes signature validation susceptible to replay across different chains.

Moreover, it is also possible to replay `sig1` in `sig2` and/or `sig3`. By doing so, we can readily replay one valid signature to pass the required threshold and execute the transaction.

Recommendation Add the `EIP-712 domainSeparator` with the `chainID` into calculation. Also, prevent the same signature from being replayed.

Status The issue has been addressed by the following commits: `09d1f0b` and `b936867`.

4.2 Denial-Of-Service Against MultisigWallet::execute()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `MultisigWallet`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

Description

In the `MultisigWallet` contract, there is a global `nonce` state that is designed to record the total number of successfully executed transactions. With that, it is used to prevent unwanted replays by being included in the computation of the generated `hash` for signature validation. However, we notice that `nonce` can be advanced by anyone even when the signature validation fails. With that, a denial-of-service situation can be introduced to block the transaction execution.

To elaborate, we list below the related `execute()` function. As mentioned in Section 4.1, this function supports three voters: Once the collected (verified) signatures is larger than the required threshold, the intended transaction will be executed. It comes to our attention that `nonce` is included in the `hash` calculation, but it can be advanced regardless of the signature validation result. Therefore, a malicious actor can simply advance to foil the signature validation, hence blocking the transaction execution.

```
37      function execute(address to, uint256 value, bytes memory data, bytes memory sig1,
            bytes memory sig2, bytes memory sig3) public payable returns (bool success) {
38          bytes32 hash = keccak256(abi.encodePacked(nonce, to, value ,data));
39          uint8 approvals;
40          approvals += _voters[hash.recover(sig1)];
41          approvals += _voters[hash.recover(sig2)];
42          approvals += _voters[hash.recover(sig3)];
43          if (approvals >= _required){
44              (success,) = to.call{value: value}(data);
45          }
46          nonce++;
47      }
```

Listing 4.2: MultisigWallet::execute()

Recommendation Move the `nonce` update inside the if-branch.

Status The issue has been addressed by the following commit: `09d1f0b`.

4.3 Reentrancy Risk In MultisigWallet::execute()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [9] exploit, and the recent `Uniswap/Lendf.Me` hack [8].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `MultisigWallet` as an example, the `execute()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 44) starts before effecting the update on internal states (line 46), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `execute()` function.

```
37    function execute(address to, uint256 value, bytes memory data, bytes memory sig1,
          bytes memory sig2, bytes memory sig3) public payable returns (bool success) {
38        bytes32 hash = keccak256(abi.encodePacked(nonce, to, value ,data));
39        uint8 approvals;
40        approvals += _voters[hash.recover(sig1)];
41        approvals += _voters[hash.recover(sig2)];
42        approvals += _voters[hash.recover(sig3)];
43        if (approvals >= _required){
44            (success,) = to.call{value: value}(data);
45        }
46        nonce++;
47    }
```

Listing 4.3: MultisigWallet :: execute()

Moreover, there is a need to validate the return boolean `success` to be `true`. Also, the `initiateVest()` routine also shares the same reentrancy risk.

Recommendation Validate the action execution result returned in `success` and add the `nonReentrant` modifier to prevent reentrancy.

Status The issue has been addressed by the following commit: `ffa42f7`.

4.4 Funds Can Be Fully Drained From Vest

- ID: PVE-004
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Vest`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The `PolkaCipher` protocol has a vesting contract that allows investors to set their own custom vesting arrangements. For each arrangement, the investor is able to specify the entire vesting period, the total amount, the beneficiary, as well as the initial amount of tokens that can be immediately available for withdrawal.

In the following, we show below the responsible `initiateVest()` function. Our analysis exposes a critical problem in this function: it firstly transfers the entire amount from the investor into this contract for custody. Afterwards, it transfers the initial amount to the intended beneficiary. However, there is a lack of validation on the given `initial`. A malicious actor can simply set `initial = _token.balanceOf(vest)` to drain all the funds in this contract.

```
62      // v.initiateVest(advisor, amount, 0, 30 days, 180 days);
63      function initiateVest(address owner, uint256 amount, uint256 initial, uint256 cliff,
            uint256 linear) public virtual returns (bytes32){
64          _token.transferFrom(_msgSender(), address(this), amount);
65          bytes32 index = getVestId(owner,_nonce[owner]);
66          _vests[index]= Vest(owner, amount, block.timestamp, initial, cliff, linear, 0);
67          _vests[index].claimed = _vests[index].claimed.add(_vests[index].initial);
68          _token.transfer(_vests[index].owner, _vests[index].initial);
69          _nonce[owner]++;
70          return index;
71      }
```

Listing 4.4: Vest::initiateVest()

Recommendation Validate the given input by ensuring `initial` is always smaller than the `amount`.

Status The issue has been addressed by the following commit: 09d1f0b.

4.5 safeTransfer()/safeTransferFrom() Replacement

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Vest.sol
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the approve() 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. On its entry of approve(), there is a requirement, i.e., require(!((_value != 0) && (allowed[msg.sender][_spender] != 0))). This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling approve(_spender, 0)) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known approve()/transferFrom() race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {
200
201         // To change the approve amount you first have to reduce the addresses`
202         //  allowance to zero by calling `approve(_spender, 0)` if it is not
203         //  already 0 to mitigate the race condition described here:
204         //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
206
207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 4.5: USDT Token Contract

Because of that, a normal call to approve() with a currently non-zero allowance may fail. Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the transfer() function does not have a return value. However, the IERC20 interface has defined the transfer()

interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
62      // v.initiateVest(advisor, amount, 0, 30 days, 180 days);
63      function initiateVest(address owner, uint256 amount, uint256 initial, uint256 cliff,
            uint256 linear) public virtual returns (bytes32){
64          _token.transferFrom(_msgSender(), address(this), amount);
65          bytes32 index = getVestId(owner,_nonce[owner]);
66          _vests[index]= Vest(owner, amount, block.timestamp, initial, cliff, linear, 0);
67          _vests[index].claimed = _vests[index].claimed.add(_vests[index].initial);
68          _token.transfer(_vests[index].owner, _vests[index].initial);
69          _nonce[owner]++;
70          return index;
71      }
```

Listing 4.6: Vest::initiateVest()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

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

Status The issue has been addressed by the following commit: `ffa42f7`.

4.6 Logic Error of getClaimable()

- ID: PVE-006
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Vest`
- Category: Business Logic [5]
- CWE subcategory: CWE-561 [3]

Description

The vesting contract in `PolkaCipher` allows users to withdraw vested assets from the contract. It also provides a convenience routine `getClaimable()` to query current claimable amount. In the following, we examine this `getClaimable()` implementation.

To elaborate, we show below the `getClaimable()` routine. It implements a rather straightforward logic by firstly subtracting the `initial` from the `amount`, then multiplying the result by `timepassed`,

and next dividing by `linear`, and finally subtracting the `claimed` from it.

```
36      function getClaimable(address owner, uint256 nonce) public view virtual returns(
            uint256 tokenClaimable) {
37          bytes32 index = getVestId(owner, nonce);
38          if (block.timestamp >= _vests[index].start.add(_vests[index].cliff) && block.
                timestamp<= _vests[index].start.add(_vests[index].cliff).add(_vests[index].
                linear)) {
39              uint256 timePassed = block.timestamp.sub((_vests[index].start).add(_vests[
                    index].cliff));
40              tokenClaimable = ((_vests[index].amount-_vests[index].initial).mul(
                    timePassed).div(_vests[index].linear)).sub(_vests[index].claimed);
41          } else if (block.timestamp > _vests[index].start.add(_vests[index].cliff).add(
                _vests[index].linear)) {
42              tokenClaimable = _vests[index].amount.sub(_vests[index].claimed);
43          }
44      }
```

Listing 4.7: Vesting::getClaimable()

It comes to our attention that the `claimed` has already included the `initial`, hence effectively subtracting the `tokenClaimable` by `initial` twice. It is considered a typical logic error.

Recommendation Add the `initial` amount back before subtracting `claimed`.

Status The issue has been addressed by the following commit: `a10242f`.

4.7 Assumed Trust on Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Token.sol`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `PolkaCipher` token contract, there is an `owner` account that plays a critical role in governing and regulating the entire operation and maintenance such as minting a certain amount ($1e9 * 1e18$) to the specified `manager`.

```
58      function mint(address manager) public virtual onlyOwner{
59          _mint(manager,1e9*1e18);
60      }
```

Listing 4.8: Token::mint()

We note that the above `mint()` function allows for the `owner` to mint more tokens into circulation without being capped. We understand the need of the privileged functions for contract operation, but at the same time the extra power to the owner may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these `onlyOwner` privileges explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to `owner` explicit to `PolkaCipher` users.

Status The issue has been addressed by the following commit: `09d1f0b`.

5 | Conclusion

In this security audit, In this security audit, we have examined the design and implementation of the `PolkaCipher` token contract. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, we identified seven issues of varying severities that were promptly confirmed and fixed by the team. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

[1] MITRE. CWE-1109: Use of Same Variable for Multiple Purposes. https://cwe.mitre.org/data/definitions/1109.html.

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

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

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

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

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

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