SMART CONTRACT AUDIT REPORT

for

everPay AR

Prepared By: Yiqun Chen

PeckShield

July 5, 2021

Document Properties

Client	everPay
Title	Smart Contract Audit Report
Target	everPay AR
Version	1.0-rc
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author	Description
1.0-rc	July 5, 2021	Yiqun Chen	Release Candidate

PeckShield

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4
1.1 About everPay AR 4
1.2 About PeckShield 5
1.3 Methodology 5
1.4 Disclaimer 7

2 Findings 8
2.1 Summary 8
2.2 Key Findings 9

3 ERC20 Compliance Checks 10

4 Detailed Results 13
4.1 Trust Issue of Admin Keys 13
4.2 Two-Step Transfer Of Privileged Account Ownership 14

5 Conclusion 16

References 17

PeckShield

1 Introduction

Given the opportunity to review the design document and related source code of the everPay AR 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 everPay AR

Arweave is a standard ERC20-compliant token governed and regulated by a multi-sig smart contract named everPay. As an ERC20-compliant token, Arweave allows users to transfer their assets by themselves, or by authorized spenders. Also, as the owner of this token contract, the everPay smart contract is able to mint tokens infinitely without being capped. However, before the mint action is executed, multiple signers should review and agree on it, or the action fails.

The basic information of everPay AR is as follows:

Table 1.1: Basic Information of everPay AR

Item	Description
Name	everPay
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	July 5, 2021

In the following, we show the audited contract code deployed at the Ethereum blockchain with the following address:

- https://kovan.etherscan.io/address/0xcc9141efa8c20c7df0778748255b1487957811be#contracts

1.2 About PeckShield

PeckShield Inc. [6] 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 [5]:

- 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 1.2: Vulnerability Severity Classification

Impact			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	*Medium*	*Low*
		Likelihood	

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 1.3: 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

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.

PeckShield

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `everPay AR` token contract. 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	0	
High	0	
Medium	1	🟦
Low	1	🟦
Informational	0	
Total	2	

PeckShield

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 medium-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key everPay AR Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue of Admin Roles	Security Features	Confirmed
PVE-002	Informational	Two-Step Transfer Of Privileged Ownership	Coding Practices	Confirmed

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

PeckShield

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 everPay AR. 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	✓

PeckShield

4 Detailed Results

4.1 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: AR
- Category: Security Features [2]
- CWE subcategory: CWE-287 [1]

Description

In the AR contract, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the entire operation and maintenance (e.g., minting tokens). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged access in current contract.

To elaborate, we show below the mint() function from the AR contract. This function allows the owner account to mint more tokens into circulation without being capped.

```
 function mint(address to, uint256 amount) onlyOwner public {
     _mint(to, amount);
   }
```

Listing 4.1: AR::mint()

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, it is worrisome if the owner is not governed by a DAO-like structure. The discussion with the team has confirmed that the owner will be managed by a multi-sig account. In this case, we still highly recommend making this onlyOwner privilege explicit or raising necessary awareness among contract users.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks.

Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance. Besides, make this extra privilege granted to `owner` explicit to everPay AR users.

Status This issue has been confirmed by the team.

4.2 Two-Step Transfer Of Privileged Account Ownership

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AR`
- Category: Coding Practices [3]
- CWE subcategory: CWE-561 [4]

Description

The `AR` contract implements a rather basic access control mechanism that allows a privileged account, i.e., `owner`, to be granted exclusive access to a typically sensitive function (i.e., `mint()`). Because of the privileged access and the implications of this sensitive function, the `owner` account is essential for the protocol-level safety and operation. In the following, we elaborate with the `owner` account.

```
    function transferOwnership(address newOwner) public onlyOwner {
        require(
            newOwner != address(0),
            "Ownable: new owner is the zero address"
        );
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
```

Listing 4.2: `AR::transferOwnership()`

The current implementation provides a specific function, i.e., `transferOwnership()`, to allow for possible `owner` updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the `newOwner` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `newOwner` is provided, the contract owner may be forever lost, which might be devastating for `AR` operation and maintenance.

As a common best practice, instead of achieving the `owner` update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the `owner` update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract `owner` to an uncontrolled address. In other words, this two-step procedure ensures that a `owner` public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the `owner` transfer process.

Recommendation Implement a two-step approach for `owner` update: `transferOwnership()` and `acceptOwnership()`.

Status This issue has been confirmed by the team.

PeckShield

5 Conclusion

In this security audit, we have examined the design and implementation of the `everPay AR` 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, although no critical or high level vulnerabilities were discovered, we identified two issues that were promptly confirmed and addressed 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.

PeckShield

References

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

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

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

[4] MITRE. CWE CATEGORY: Two-Step Transfer Of Privileged Account Ownership. https://cwe.mitre.org/data/definitions/561.html.

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

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

PeckShield