SMART CONTRACT AUDIT REPORT

for

DRAM Stablecoin

Prepared By: Xiaomi Huang

PeckShield

August 26, 2023

Document Properties

Client	DRAM
Title	Smart Contract Audit Report
Target	DRAM
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	August 26, 2023	Xuxian Jiang	Final Release
1.0-rc	August 14, 2023	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the DRAM 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 the smart contract can be further improved due to the presence of certain issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About DRAM

DRAM is an ERC-20 token issued by DRAM Trust and it is tied directly to the value of AED (United Arab Emirates Dirham), fully backed 1:3.625 with AED held for the benefit of the DRAM token holders. The ratio of DRAM to AED reflects the USD:AED peg that has been in place since 1997. The DRAM is a stable coin that combines the creditworthiness and stability of the Dirham with the technology and efficiency advantages of digital assets. The issuance, redemption and reserve management are overseen by DRAM Trust, regulated in Hong Kong, and managed by a team of seasoned bankers and technologists with hundreds of years of combined experience. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of DRAM

Item	Description
Name	DRAM
Type	ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 26, 2023

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

- https://github.com/Calculum-Companies-Representation/DRAM-contracts.git (291600c)

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

- https://github.com/Calculum-Companies-Representation/DRAM-contracts.git (70fc1cd)

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.

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

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

Likelihood

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 `DRAM` 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	

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. While there is no critical or high severity issue, the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 1 low-severity vulnerability.

Table 2.1: Key DRAM Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Initialization Logic in Dram::initialize()	Coding Practices	Resolved
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate the above issue(s), 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.

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 `DRAM` token contract. 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-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	✓
Approval() event	Is emitted on any successful call to approve()	✓

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, 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 Improved Initialization Logic in Dram::initialize()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Dram`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

The `Dram` contract allows for lazy contract initialization, i.e., the initialization does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the `initializer()` and `onlyInitializing()` modifiers. The `initializer()` protects an initializer function from being invoked twice, and the `onlyInitializing()` modifier protects an initialization function so that it can only be invoked by functions with the `initializer()` modifier, directly or indirectly. While examining the current initialization sequence, we notice the implementation may be improved.

To elaborate, we show below the code snippet of the `Dram::initialize()` routine. As the name indicates, it is an initialization function for the `Dram` contract. This `initialize()` function is protected by the `initializer()` and it further invokes the subcalls to `__ERC20_init`, `__Pausable_init`, `__ERC20Permit_init`, and `__DramAccessControl_init` (lines $36-39$). It comes to our attention that the `initializer()` should also make other subcalls to `__DramMintable_init()/__DramFreezable_init()`.

```
35    function initialize(address admin) public initializer {
36        __ERC20_init("Dram", "DRM");
37        __Pausable_init();
38        __ERC20Permit_init("Dram");
39        __DramAccessControl_init(admin);
40    }
```

Listing 4.1: `Dram::initialize()`

Similarly, the initialization logic for the `DramTimelockController` should also be extended to call `__AccessControl_init()`.

Recommendation Improve existing initialization sequence with additional subcalls.

Status This issue has been fixed in this commit: `ceb4661`.

4.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `Dram` implementation, there are privileged accounts that play a critical role in governing and regulating the token-wide operations (e.g., mint more tokens into circulation and pause the token). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `Dram` contract as an example and show the representative functions potentially affected by these privileged accounts.

```
    function pause() external onlyRoleOrAdmin(REGULATORY_MANAGER_ROLE) {
        _pause();
    }

    /**
     * @notice Resumes the smart contract.
     */
    function unpause() external onlyRoleOrAdmin(REGULATORY_MANAGER_ROLE) {
        _unpause();
    }

    /**
     * @notice Freezes an account. A freezed account can't send any transaction related
         to
     * transferring tokens.
     * @dev Protected by onlyRoleOrAdmin, only admin and regulatory manager can call the
         function.
     * @param account Account to be freezed
     */
    function freeze(
        address account
    ) external onlyRoleOrAdmin(REGULATORY_MANAGER_ROLE) {
        _freeze(account);
    }
```

Listing 4.2: Example Privileged Operations in `Dram`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. And it is worrisome since the privileged account is a hardcoded plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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.

Status This issue has been confirmed and mitigated with a planned multisig or timelock.

5 Conclusion

In this security audit, we have examined the DRAM token design and implementation. 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 level vulnerabilities were discovered, we identified three issues that need to be promptly addressed. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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