SMART CONTRACT AUDIT REPORT

for

ERC20H

Prepared By: Xiaomi Huang

PeckShield

May 31, 2025

Document Properties

Client	Drip.Trade
Title	Smart Contract Audit Report
Target	ERC20H
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 31, 2025	Xuxian Jiang	Final Release
1.0-rc	May 30, 2025	Xuxian Jiang	Release Candidate #1

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 ERC20H 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/ERC721-compliance, security, or performance. This document outlines our audit results.

1.1 About ERC20H

ERC20H is a new token primitive that redefines the boundary between fungible and non-fungible assets. The H stands for Hybrid – reflecting its core design: ERC20 tokens can be locked and transmuted into transferable NFTs, enabling new forms of value containment, composability, and social coordination. Locked tokens remain visible in balances but become non-transferable unless bonded to NFTs, which are minted automatically based on configurable thresholds and tiers. These NFTs carry the underlying tokens with them and must be burned to release the bonded assets – creating a new programmable liquidity primitive. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of ERC20H

Item	Description
Name	ERC20H
Type	Token Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 31, 2025

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

- https://github.com/driptrade/ERC20H.git (4403a96)

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

- https://github.com/driptrade/ERC20H.git (60d5284)

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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/ERC721 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20/ERC721 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

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

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/ERC721 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 ERC20H 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/ERC721-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	1	■
Total	4	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20/ERC721 specification and other known best practices, and validate its compatibility with other similar ERC20/ERC721 tokens and current DeFi protocols. The detailed ERC20/ERC721 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, there exists an ERC20/ERC721 compliance issue 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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key ERC20H Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper setUnlockCooldown() Logic in ERC20HReleasable	Security Features	Resolved
PVE-002	Informational	Simplified _getMintableTokenIds() Logic in ERC20HMirror	Coding Practices	Resolved
PVE-003	Low	Revisited getMintableNumberOfTokens() Logic in ERC20HMirror	Business Logic	Resolved
PVE-004	Low	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/ERC721 Compliance

The ERC20/ERC721 specifications define 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/ERC721-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).

3.1 ERC20 Compliance

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 in the audited `ERC20H` 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 ERC20 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	—
Whitelistable	The token contract allows the owner or privileged users to whitelist a specific address such that only token transfers and other operations related to that address are allowed	—
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	—

3.2 ERC721 Compliance

The ERC721 standard for non-fungible tokens, also known as deeds. Inspired by the ERC20 token standard, the ERC721 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 ERC721-compliant. Naturally, we examine the list of necessary API functions defined by the ERC721 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.4: Basic View-Only Functions Defined in The ERC721 Specification

Item	Description	Status
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
ownerOf()	Is declared as a public view function	✓
	Returns the address of the owner of the NFT	✓
getApproved()	Is declared as a public view function	✓
	Reverts while ‘_tokenId‘ does not exist	✓
	Returns the approved address for this NFT	✓
isApprovedForAll()	Is declared as a public view function	✓
	Returns a boolean value which check ‘_operator‘ is an approved operator	✓

Our analysis shows that the balanceOf() function is defined to be ERC20-compliant. Thus, this specific function does not count all NFTs assigned to an owner. And there is no other ERC721 inconsistency or incompatibility issue found in the audited ERC1000 token contract. In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.4) and key state-changing functions (Table 3.5) according to the widely-adopted ERC721 specification.

Table 3.5: Key State-Changing Functions Defined in The ERC721 Specification

Item	Description	Status
safeTransferFrom()	Is declared as a public function	✓
	Reverts while ‘to‘ refers to a smart contract and not implement IERC721Receiver-onERC721Received	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while ‘_tokenId‘ is not a valid NFT	✓
	Reverts while ‘_from‘ is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
transferFrom()	Is declared as a public function	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Reverts while ‘_tokenId‘ is not a valid NFT	✓
	Reverts while ‘_from‘ is not the current owner	✓
	Reverts while transferring to zero address	✓
	Emits Transfer() event when tokens are transferred successfully	✓
approve()	Is declared as a public function	✓
	Reverts unless ‘msg.sender‘ is the current owner, an authorized operator, or the approved address for this NFT	✓
	Emits Approval() event when tokens are approved successfully	✓
setApprovalForAll()	Is declared as a public function	✓
	Reverts while not approving to caller	✓
	Emits ApprovalForAll() event when tokens are approved successfully	✓
Transfer() event	Is emitted when tokens are transferred	✓
Approval() event	Is emitted on any successful call to approve()	✓
ApprovalForAll() event	Is emitted on any successful call to setApprovalForAll()	✓

4 | Detailed Results

4.1 Improper setUnlockCooldown() Logic in ERC20HReleasable

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: ERC20HReleasable
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The ERC20H protocol is innovative in helping foster community nucleation through locking and bonding fungible tokens into NFTs. It also allows tokens that are locked but not bonded to an NFT may have an unlock cooldown set. Once a user begins the unlocking process, the user must wait a predetermined amount of time before they can release the tokens and make them available for transfer.

While reviewing the logic to set up the cooldown duration, we notice the setter setUnlockCooldown() function has two issues. The first one is the lack of access control since current implementation allows for any one to update this important paramter (line 30). The second one is the implicit assumption, i.e., the new unlock cooldown duration should be monotonically increasing. And this implicit assumption is not currently being enforced (line 612). In other words, we need to check the following statement, i.e., `require(unlockCooldown_ > _unlockCooldown);`, before applying the new unlock cooldown duration.

```
30      function setUnlockCooldown(uint96 unlockCooldown_) external virtual {
31          _setUnlockCooldown(unlockCooldown_);
32      }
```

Listing 4.1: ERC20HReleasable::setUnlockCooldown()

```
611     function _setUnlockCooldown(uint96 unlockCooldown_) internal virtual {
612         _unlockCooldown = unlockCooldown_;
613     }
```

Listing 4.2: ERC20H::_setUnlockCooldown()

Recommendation Revist the above routine to validate the caller and the new cooldown duration parameter.

Status This issue has been resolved in the following commits: 26769aa, 67713af, and e4ef413.

4.2 Simplified _getMintableTokenIds() Logic in ERC20HMirror

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: ERC1000
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

When a user locks up her tokens, the locked tokens will automatically be used to mint NFTs. If minted, the NFTs are bonded with the fungible token. In the process of examining the logic to compute the mintable NFTs, we notice current implementation can be optimized.

```
    // if there are any unused slots in the array, clean them up
    if (tokensAdded > 0 && tokensAdded < numTokens) {
        uint256 unusedSlots;
        unchecked {
            unusedSlots = numTokens - tokensAdded;
        }
        assembly {
            mstore(tokenIds, sub(mload(tokenIds), unusedSlots))
        }
    }
```

Listing 4.3: ERC20HMirror::_getMintableTokenIds()

To elaborate, we show above the code snippet from the related _getMintableTokenIds() routine. As the name indicates, this routine is used to compute the mintable NFTs from the given bonded amount. It comes to our attention that the mintable NFTs are saved in the return array of tokenIds . This array is initialized with the maximum entries of numTokens. If the return array is not fully populated, current implementation reduces the array size by unusedSlots. Note it can be simplified as mstore(tokenIds, tokensAdded) without the need of introducing additional unusedSlots variable and thus avoiding associated computation.

Recommendation Simplify the above-mentioned routine by removing unnecessary variable and associated computation.

Status This issue has been resolved in the following commit: f6568d4.

4.3 Revisited getMintableNumberOfTokens() Logic in ERC20HMirror

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC20HMirror`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In `ERC20H`, once fungible tokens are bonded to an `NFT`, they can only be retrieved by unbonding. Unbonding will burn the `NFT` and unlock its fungible tokens. By design, if an `NFT` tier has a max supply, any burnt `NFT` from that tier will be permanent. While reviewing the permanency of burnt `NFT`, we notice an issue in the calculation of mintable `NFTs` from the given fungible tokens.

To elaborate, we show below the implementation of the related `getMintableNumberOfTokens()` routine. It comes to our attention that the remaining supply of `NFTs` is computed as `remainingSupply = uint256(tierInfo.maxSupply - tierInfo.totalSupply)` (line 751), which should be revised as `remainingSupply = uint256(tierInfo.maxSupply - tierInfo.nextTokenIdSuffix)`. Note this issue also affects another routine in the same contract, i.e., `_getMintableTokenIds()`).

```
729     function _getMintableNumberOfTokens(uint256 backing) internal view virtual returns (
            uint256, uint256) {
730         uint256 numActiveTiers = _activeTiers.length;
731         uint256 cur;
732         uint256 numTokens;
733         uint256 remaining = backing;

735         while (cur < numActiveTiers && backing > 0) {
736             uint16 tierId;
737             unchecked { tierId = _activeTiers[cur]; }

739             TierInfo storage tierInfo = _getTierUnsafe(tierId);

741             uint256 tierUnits = uint256(tierInfo.units);

743             if (remaining < tierUnits) {
744                 unchecked { cur += 1; }
745                 continue; // move on to the next tier
746             }

748             uint256 remainingSupply;
749             uint256 multiple;
750             unchecked {
751                 remainingSupply = uint256(tierInfo.maxSupply - tierInfo.totalSupply);
752                 multiple = remaining / tierUnits;
```

```
            }
            if (multiple > remainingSupply) {
                multiple = remainingSupply;
            }

            // there are tokens that can be minted from this tier
            if (multiple > 0) {
                unchecked {
                    remaining -= multiple * tierUnits;
                    // backing decreases per iteration. sum of quotients <= sum of dividends
                    numTokens += multiple;
                }
            }

            unchecked { cur += 1; }
        }

        return (numTokens, backing - remaining);
    }
```

Listing 4.4: ERC20HMirror::getMintableNumberOfTokens()

Recommendation Improve the above-mentioned routines to ensure the remaining supply is properly computed.

Status This issue has been resolved in the following commit: c2e7a7b.

4.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the ERC20H/ERC20HMirror token contract, there is a privileged owner account that plays a critical role in governing and regulating the contract-wide operations (e.g., manage and activate tiers). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contract.

```
    function addTiers(AddTierParam[] calldata tierParams) external virtual onlyOwner {
        _addTiers(tierParams);
    }

    function removeTier(uint16 tierId) external virtual onlyOwner {
        _removeTier(tierId);
    }

    function setActiveTiers(uint16[] calldata tierIds) external virtual onlyOwner {
        _clearActiveTiers();
        _setActiveTiers(tierIds);
    }
```

Listing 4.5: Example Privileged Operations in ERC20HMirror

```
    /**
     * @dev Allows contract owner to set the mirror contract.
     */
    function setMirror(address mirror_) external virtual onlyOwner {
        _setMirror(mirror_);
    }
```

Listing 4.6: The Privileged Operation in ERC20H

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it would be worrisome if the privileged account is a 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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.

Status This issue has been mitigated with the use of a multi-sig account to hold the admin account.

5 Conclusion

In this security audit, we have examined the ERC20H token design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC20/ERC721 specification and other known ERC20/ERC721 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, there are no critical level vulnerabilities discovered and other identified issues are promptly addressed.

References

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

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

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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