hexens x AnchorVaultCoin

Security Review Report for AnchorVaultCoin

July 2026

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for AnchorVault protocol, a non-custodial multi-asset vault.

Our security assessment was a full review of the AnchorVaultCoin contract code, spanning a total of 1 week.

During our review, we did not identify any major severity vulnerabilities.

We did identify some minor severity vulnerabilities and code optimizations.

All reported issues were fixed by the development team and subsequently verified by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

▪ Review Led by

Kasper Zwijsen, Head of Audits

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/anchorvaultcoin-hash/anchor-vault-v45/blob/0827e11860abc0d1b27b86afb8f5cd9bb73db094/src/AnchorVaultV45.sol

The issues described in this report were fixed in the following commit:

🔗 https://github.com/anchorvaultcoin-hash/anchor-vault-v45/blob/6fead3f2f325a1f57a23efd91cfe5f76c7727528/src/AnchorVaultCoin.sol

▪ Changelog

20 July 2026	Audit start
31 July 2026	Initial report
3 August 2026	Revision received
10 August 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

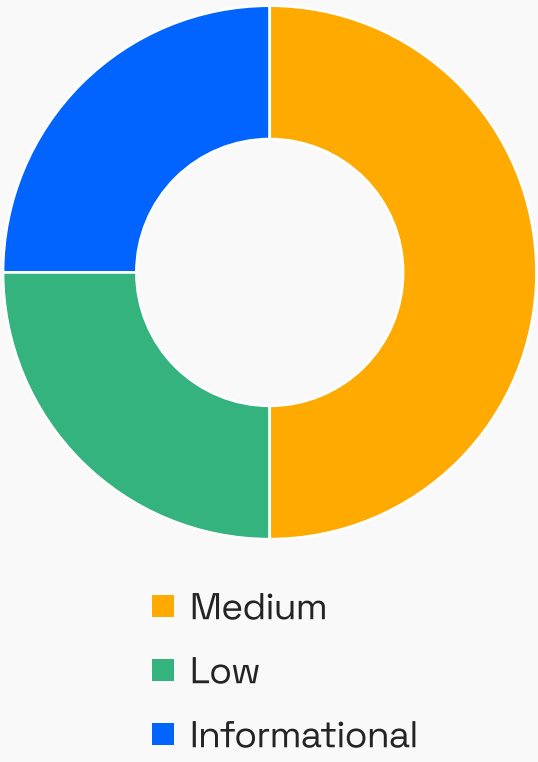
- **Issue Symbolic Codes**

Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	2
Low	1
Informational	1
Total:	**4**



6. Weaknesses

This section contains the list of discovered weaknesses.

ANCV1-1 | Public proof validation allows proof token griefing

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

src/AnchorVaultV45.sol:_closeTransfer#L735-L740

Description:

The secure transfer flow lets a vault owner lock a vault for a recipient, who can then confirm or reject the transfer. **initSecureTransfer** marks the source vault as pending, and **confirmSecureTransfer** moves it to the recipient.

Each pending transfer should have exclusive control over its source vault. Closing an older transfer must not unlock a vault that has since been locked by a newer transfer, and confirmation should only succeed while the expected source vault still exists.

However, a recipient conflict unlocks the source vault but leaves the transfer record in **CONFLICT** status. Such a record can still be passed to **cancelSecureTransfer** or **reclaimExpiredTransfer**. Both functions call **_closeTransfer**, which blindly sets the shared vault's status to active without checking whether a newer transfer now owns the lock:

```
function _closeTransfer(uint256 /*transferId*/, SecureTransfer storage st, uint8 newStatus) internal {
    address token = vaults[st.from][st.vaultId].token;
    vaults[st.from][st.vaultId].status = 0;
    pendingIncomingTransfer[st.to][token] = 0;
    st.status = newStatus;
}
```

This lets a malicious sender unlock and quick-transfer the vault while a newer secure transfer still appears pending. If the new recipient confirms, the contract reads the deleted source storage and creates a zero-value vault whose token is **address(0)**. It also clears the incoming slot for **address(0)** instead of the real token, permanently blocking the recipient from receiving future vault transfers for that token.

An attack can happen as follows:

1. The attacker starts a secure transfer to a helper, then makes the helper open a vault for the same token and confirm. This triggers the conflict branch, unlocks the source vault, and leaves the old record in **CONFLICT** status.
2. The attacker starts a new secure transfer of the same vault to the victim.
3. The attacker cancels the old conflict record. **_closeTransfer** unlocks the vault even though it now backs the victim's pending transfer.
4. The attacker quick-transfers the unlocked vault to another address, deleting the original source storage.
5. The victim confirms the pending transfer. The call succeeds but delivers a zero-value, zero-token vault, and the victim's incoming slot for the real token remains permanently occupied.

Remediation:

Bind each vault lock to a specific transfer ID and only unlock the vault or clear an incoming slot when it still belongs to that transfer. Confirmation should also verify that the source vault exists, remains locked by the same transfer, and contains the expected token and value.

ANCV1-2 | Griefing Attack Allows Unauthorized Vault Creation for Arbitrary Users

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

src/AnchorVaultV45.sol#L568-L598

Description:

Function **transferVault()** is designed for the sender to set up a new vault under the address **to** and transfer the vault token to the new vault. It requires a signature from the sender's vault **mainAuthKey** to perform the action. However, it does not require any signature from the **to** address, which allows anyone to create a vault for an arbitrary address with sender-chosen **mainAuthKey** and **recoverAuthKey**. This creates a flaw where an attacker can front-run a user and create a vault on their behalf with arbitrary keys.

```
function transferVault(
    uint256 vid, address to, address newMainKey, address newRecoveryKey,
    uint256 deadline, bytes calldata sig
) external nonReentrant whenNotPaused vaultExists(msg.sender, vid) {
    ...

    bytes32 sh = keccak256(abi.encode(TRANSFER_TYPEHASH, msg.sender, vid, to, newMainKey,
newRecoveryKey, v.nonce, deadline));
    _checkSig(v, sh, deadline, sig, v.mainAuthKey, true);

    ...

    uint256 newId = _createReceivedVault(to, token, net, v.level, newMainKey, newRecoveryKey, v.name);

    ...
}
```

Consider the following scenario where a user with address **0xA** wants to open a vault for the USDC token and then deposit 1M USDC into that vault. The steps that **0xA** performs are:

1. Get the vault ID by calculating **vid = userVaultCount[0xA] + 1**.
2. Trigger **openVault()** without checking whether the transaction has succeeded.
3. Deposit 1M USDC into the vault with ID **vid** obtained in step 1.

An attacker can inspect these actions and front-run by executing a malicious action between steps 1 and 2. Specifically, the attacker calls **transferVault()** with:

- to = 0xA
- newMainKey = 0xAttackerMain
- newRecoveryAuthKey = 0xAttackerRecovery

By specifying arbitrary **newMainKey** and **newRecoveryAuthKey** values, the attacker can create a vault on behalf of the user using attacker-controlled keys. Because the vault id that **transferVault()** function creates for the **to** address is still **vid** getting from step 1, hence the 1M USDC is still deposited to the vault. As a result, the attacker forces user **0xA** into a state where they cannot perform any vault operations (because they do not have the required authorization keys) except **panicWithdraw()**, which incurs a 20% penalty on their funds (200k USDC in this example).

Remediation:

Consider requiring the signature from the address **to** in the function **transferVault()**.

ANCV1-4 | Tier-based deposit fees can be reduced by withdrawing and reopening

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Description:

Deposits made through depositToVault are charged according to the vault tier: 50 bps for SAFE, 150 bps for VAULT, and 200 bps for FORTRESS. In contrast, **openVault** charges a fixed 20 bps regardless of the selected tier.

```
uint256 public constant SAFE_DEPOSIT_FEE_BPS = 50;
uint256 public constant VAULT_DEPOSIT_FEE_BPS = 150;
uint256 public constant FORTRESS_DEPOSIT_FEE_BPS = 200;

uint256 public constant OPEN_VAULT_FEE_BPS = 20;
uint256 public constant WITHDRAW_FEE_BPS = 50;
```

A full withdrawal clears the active vault slot, allowing the owner to combine the withdrawn amount with new funds and reopen a vault instead of topping up the existing one. When the new deposit is sufficiently large compared with the existing principal, this route collects fewer fees for every tier. The difference is greatest for **FORTRESS** because its direct top-up rate is 200 bps.

For example, with **100,000 ANCR** in a **FORTRESS** and a planned **100,000 ANCR** top-up, a direct deposit charges **2,000 ANCR**. Withdrawing and reopening charges **899 ANCR** in total **500 ANCR** for the withdrawal and **399 ANCR** for reopening reducing the collected fees by **1,101 ANCR**.

Remediation:

- Apply the selected vault tier's deposit fee when calculating fees in **openVault**.
- If the discounted opening rate is intentional, cap its eligible amount and preserve eligibility across full withdrawals and subsequent openings.

ANCV1-3 | The minimum deposit amount is checked twice when depositing into a vault

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/AnchorVaultV45.sol#L520
src/AnchorVaultV45.sol#L446

Description:

In the **depositToVault()** function, the minimum deposit requirement is checked twice:

- The first check validates the raw input **amount** at line 520.
- The second check validates the actual amount received (**net**) at line 524.

```
function depositToVault(uint256 vid, uint256 amount)
    external nonReentrant whenNotPaused vaultExists(msg.sender, vid)
{
    ...
    if (amount < minDep) revert DepositBelowMinimum();
    ...
    if (net < minDep) revert DepositBelowMinimum();
    ...
}
```

Since **net** is always less than or equal to the input **amount** (due to protocol fees and potential Fee-on-Transfer token deductions), the first check is redundant. The actual amount credited to the vault is what ultimately matters for enforcing the minimum deposit requirement.

The same issue exists in the **openVault()** function.

Remediation:

Consider removing the minimum deposit check on the input amount and enforcing the requirement only against the actual amount received (net). This simplifies the logic and ensures the validation is based on the amount that is effectively deposited into the vault.

hexens x  AnchorVaultCoin