AstraSec

FriendSpace Security Audit Report

December 23, 2025

Contents

1 Introduction

1.1 About FriendSpace

FriendSpace is a social token platform on Base that lets creators issue tokenized social shares via bonding curves. Users can buy and sell creator shares, creating a marketplace with economic incentives.The protocol uses an ERC-1155 token contract with bonding curve pricing where price increases with supply. Each creator has a unique token ID representing their shares. The fee structure includes dev fees, creator fees, and trading pool fees. All contracts are upgradeable using the UUPS proxy pattern for future upgrades.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/FriendDotSpace/contracts/tree/main/src
- Commit ID: 6f32341dea60aefb06a18a9669f02caf463a663e

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/FriendDotSpace/contracts/tree/main/src
- Commit ID: bf38391b48888b44388209d8c144438dc28e63cb

1.3 Revision History

Version	Date	Description
v1.0	December 23, 2025	Initial Audit

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the FriendSpace protocol. Throughout this audit, we identified a total of 4 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	–	–	–	–
High	1	–	–	1
Medium	2	1	–	1
Low	1	–	–	1
Informational	–	–	–	–
Total	4	1	–	3

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~H-1~~	Accommodation of Non-ERC20 Compliant Tokens
~~M-1~~	Denial-of-Service Attack on Reward Distribution
M-2	Potential Risks Associated with Centralization
~~L-2~~	Improved Bridge giveAmount Validation in _dispatch()

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Acknowledged
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

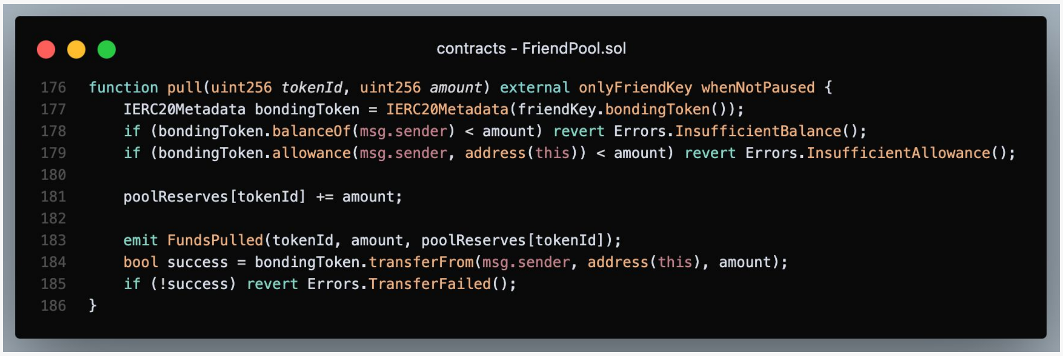
3.3 Vulnerability Details

3.3.1 [H-1] Accommodation of Non-ERC20 Compliant Tokens

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Multiple Files	Incompatibility	High	Medium	Addressed

The protocol uses direct transferFrom() and approve() calls with boolean return value checks, which may be incompatible with non-ERC20 compliant tokens. When transferFrom() or approve() is called on tokens like USDT (on certain chains), these functions do not return a boolean value. The contract expects a boolean return and checks it, causing the transaction to revert when interacting with such tokens. What's more, approve() calls may fail if there is an existing non-zero allowance, as some tokens require resetting the allowance to zero before setting a new value.

The contract will revert when interacting with non-ERC20 compliant tokens, making it incompatible with widely-used tokens like USDT on various chains. This can result in complete loss of functionality for critical operations such as token transfers and fee approvals, potentially locking user funds if the bonding token is non-compliant. The issue affects all token transfer and approval operations throughout the protocol.


contracts - FriendPool.sol

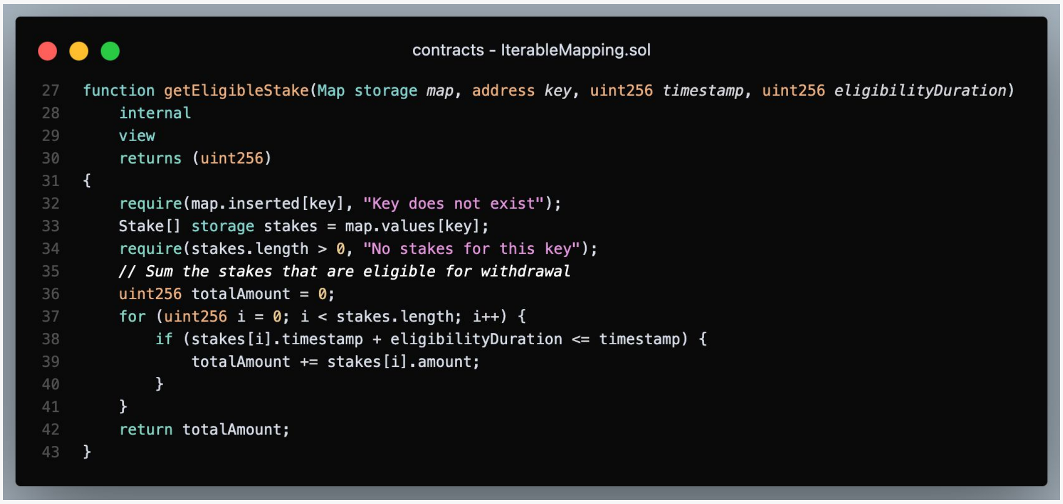
```
function pull(uint256 tokenId, uint256 amount) external onlyFriendKey whenNotPaused {
    IERC20Metadata bondingToken = IERC20Metadata(friendKey.bondingToken());
    if (bondingToken.balanceOf(msg.sender) < amount) revert Errors.InsufficientBalance();
    if (bondingToken.allowance(msg.sender, address(this)) < amount) revert Errors.InsufficientAllowance();

    poolReserves[tokenId] += amount;

    emit FundsPulled(tokenId, amount, poolReserves[tokenId]);
    bool success = bondingToken.transferFrom(msg.sender, address(this), amount);
    if (!success) revert Errors.TransferFailed();
}
```

Remediation Replace all direct transferFrom() calls with safeTransferFrom() and all approve() calls with forceApprove() from OpenZeppelin's SafeERC20 library.

3.3.2 [M-1] Denial-of-Service Attack on Reward Distribution

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
FriendStake.sol	Denial-of-Service	Medium	Medium	Addressed

The distributeRewards() function iterates through users sequentially in the stakedBalances mapping and calls getEligibleStake() for each user to determine their eligible stake amount. However, getEligibleStake() contains a strict requirement that stakes.length > 0 (line 34), which causes the function to revert if a user has no stakes. A malicious user can exploit this by staking tokens to gain a position in the stakedBalances mapping, then unstaking all their tokens. If the user remains in the mapping but has an empty stakes array, any subsequent call to distributeRewards() will revert when it attempts to process that user, effectively denying service to all other legitimate users who should receive rewards. This creates a single point of failure where one malicious actor can permanently block the entire reward distribution process.



```
27  function getEligibleStake(Map storage map, address key, uint256 timestamp, uint256 eligibilityDuration)
28      internal
29      view
30      returns (uint256)
31  {
32      require(map.inserted[key], "Key does not exist");
33      Stake[] storage stakes = map.values[key];
34      require(stakes.length > 0, "No stakes for this key");
35      // Sum the stakes that are eligible for withdrawal
36      uint256 totalAmount = 0;
37      for (uint256 i = 0; i < stakes.length; i++) {
38          if (stakes[i].timestamp + eligibilityDuration <= timestamp) {
39              totalAmount += stakes[i].amount;
40          }
41      }
42      return totalAmount;
43  }
```

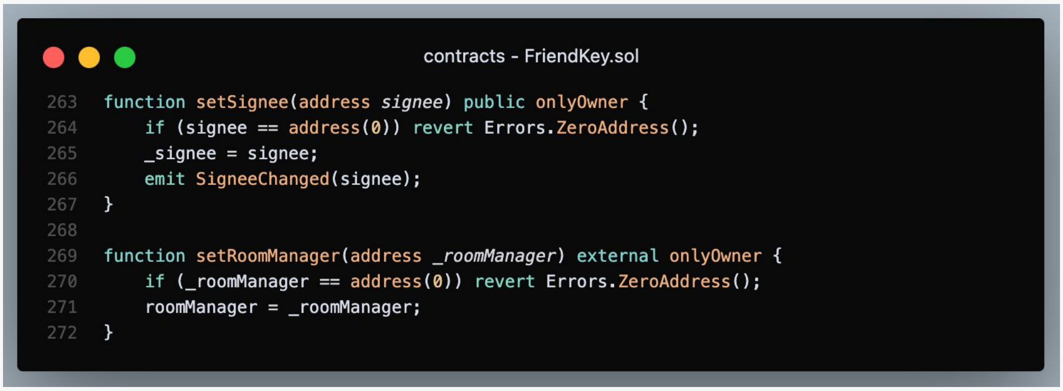
Remediation Modify distributeRewards() to handle users with empty stakes gracefully, or update getEligibleStake() to return 0 instead of reverting when there are no stakes. It's also doable to explicitly check if the user has unstaked all their tokens and remove them from the mapping in _unstake().

3.3.3 [M-2] Potential Risks Associated with Centralization

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	Medium	Low	Acknowledged

The protocol grants extensive protocol-wide control to a privileged owner account, introducing significant centralization risks. Specifically, this account can authorize contract upgrades, set critical addresses (friendKey, authority, signee, room manager), control pause/unpause functionality, configure all fee destinations and fee rates (dev fees, creator fees, trading pool fees, performance fees, social fees, dispatch fee), modify bonding curve divisors, and control the signee address used for creator registration signatures.

The extensive owner privileges create significant centralization risks, including potential fund loss, protocol manipulation, and complete control over protocol operations. A compromised or malicious owner could execute harmful changes immediately without safeguards or community oversight.


contracts - FriendKey.sol

```
function setSignee(address signee) public onlyOwner {
    if (signee == address(0)) revert Errors.ZeroAddress();
    _signee = signee;
    emit SigneeChanged(signee);
}

function setRoomManager(address _roomManager) external onlyOwner {
    if (_roomManager == address(0)) revert Errors.ZeroAddress();
    roomManager = _roomManager;
}
```

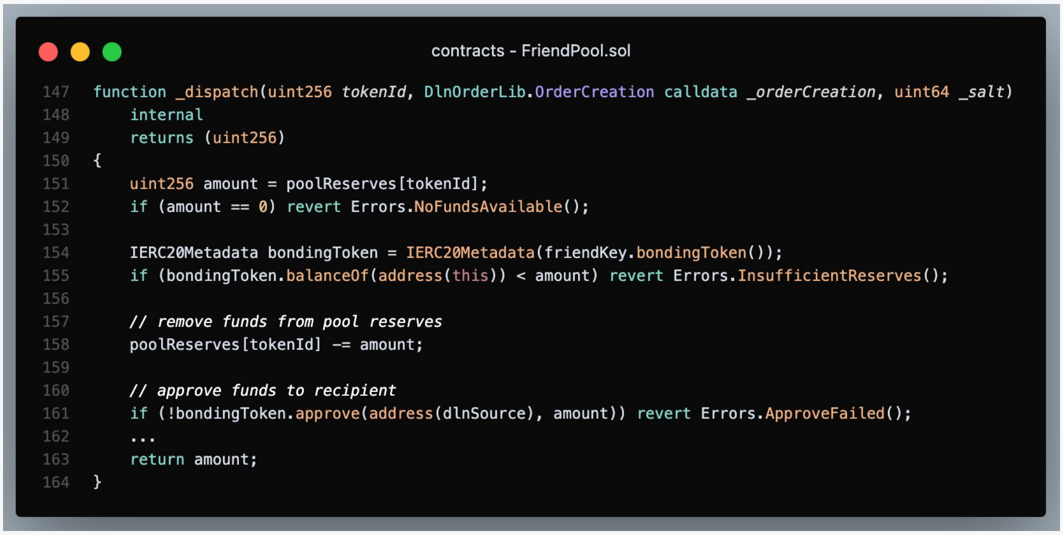
Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged accounts. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been acknowledged by the team.

3.3.4 [L-1] Improved Bridge giveAmount Validation in _dispatch()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
FriendPool.sol	Input Validation	Low	Low	Addressed

The _dispatch() function retrieves the amount from poolReserves[tokenId], reduces the reserves by that amount, and then creates a DLN order using the _orderCreation parameter which contains a giveAmount field. However, the function does not validate that the amount being dispatched from reserves matches the giveAmount specified in the _orderCreation parameter. This creates a potential mismatch where the contract reduces reserves by one amount but creates an order with a different giveAmount, leading to accounting inconsistencies and potential discrepancies between the actual funds being transferred and the order parameter


contracts - FriendPool.sol

```
function _dispatch(uint256 tokenId, DlnOrderLib.OrderCreation calldata _orderCreation, uint64 _salt)
    internal
    returns (uint256)
{
    uint256 amount = poolReserves[tokenId];
    if (amount == 0) revert Errors.NoFundsAvailable();

    IERC20Metadata bondingToken = IERC20Metadata(friendKey.bondingToken());
    if (bondingToken.balanceOf(address(this)) < amount) revert Errors.InsufficientReserves();

    // remove funds from pool reserves
    poolReserves[tokenId] -= amount;

    // approve funds to recipient
    if (!bondingToken.approve(address(dlnSource), amount)) revert Errors.ApproveFailed();
    ...
    return amount;
}
```

Remediation Add validation to ensure that the amount being dispatched from reserves matches the giveAmount specified in the order creation parameters.

4 Appendix

4.1 About AstraSec

AstraSec is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, AstraSec maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. AstraSec's comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://x.com/AstraSecAI