Guild Audits

JANUARY, 2026

ESUSU PROTOCOL FINAL AUDIT REPORT

By Guild Audits

TABLE OF CONTENTS

DISCLAIMER

A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

EXECUTIVE SUMMARY

Guild Academy was commissioned by the Esusu Protocol team to conduct a comprehensive security audit of their smart contract infrastructure. The primary objective of this engagement was to identify potential vulnerabilities, assess the security posture of the protocol's core logic, and provide actionable recommendations to safeguard user funds and ensure protocol integrity prior to [mainnet deployment/an upcoming upgrade].
This document serves as a high-level overview of the audit scope, findings, remediation efforts, and the final security verdict concluded on 8th January, 2026 .

Commit: 208efb6f684c3c5582d2cab2ae17dee2e824d2de in main branch of code

FINDINGS

Index	Title	Severity	Status of Issues
01	[H-1] Broken Factory Upgrade Mechanism	High	Resolved
02	[H-2] Malicious Upgrade via Insufficient Timelock Delay	High	Resolved
03	[H-03] Depositors Receive Zero Interest	High	Resolved
04	[H-4] leaveGroup() will always revert because updateUserBalance() deducts from a balance that was never credited.	High	Resolved
05	[H-5] leaveGroup() emits refund event but never transfers tokens — user funds remain locked in contract	High	Resolved
06	[M-1] Excess Contribution Amounts Permanently Trapped	Medium	Resolved
07	[M-2] Emergency Withdrawal Timelock Defeats Emergency Purpose	Medium	Resolved

FINDINGS

Index	Title	Severity	Status of Issues
08	[M-3] Thrift Group Contributions Do Not Earn Yield	Medium	Resolved
09	[M-4] Factory Does Not Track Deployed Proxies.	Medium	Resolved
10	[L-1] Identical Events for Pool & PoolDataProvider Updates	Low	Resolved
11	[L-2] Deposit Timestamp Overwritten and Unused.	Low	Resolved
12	[L-3] Misleading Error Message in withdrawFromAave.	Low	Resolved

MODE OF AUDIT AND METHODOLOGIES

The mode of audit carried out in this audit process is as follows:
Manual Review: This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
Automated Testing: This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools used are Slither, Echidna and others.
Functional Testing: Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to exploit the contracts.

The methodologies for establishing severity issues:

- High Level Severity Issues
- Medium Level Severity Issues
- Low Level Severity Issues
- Informational Level Severity Issues

TYPES OF SEVERITY

Every issue in this report has been assigned a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

These are critical issues that can lead to a significant loss of funds, compromise of the contract's integrity, or core function of the contract not working. Exploitation of such vulnerabilities could result in immediate and catastrophic consequences, such as:

- Complete depletion of funds.
- Permanent denial of service (DoS) to contract functionality.
- Unauthorized access or control over the contract

Medium Severity Issues

These issues pose a significant risk but require certain preconditions or complex setups to exploit. They may not result in immediate financial loss but can degrade contract functionality or pave the way for further exploitation. Exploitation of such vulnerabilities could result in partial denial of service for certain users,leakage of sensitive data or unintended contract behavior under specific circumstances.

Low Severity Issues

These are minor issues that have a negligible impact on the contract or its users. They may affect efficiency, readability, or clarity but do not compromise security or lead to financial loss. Impacts are minor degradation in performance,confusion for developers or users interacting with the contract and low risk of exploitation with limited consequences.

Informational

These are not vulnerabilities in the strict sense but observations or suggestions for improving the contract's code quality, readability, and adherence to best practices.

There is no direct impact on the contract’s functionality or security, it is aimed at improving code standards and developer understanding.

TYPES OF ISSUES

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the audit and have been successfully fixed.

Acknowledged

Vulnerabilities that have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

REPORT OF FINDINGS

HIGH SEVERITY ISSUE

[H-01] Broken Factory Upgrade Mechanism

- **Description**

The *MiniSafeFactoryUpgradeable* contract includes functions intended to upgrade deployed instances of the protocol (*upgradeSpecificContract* and *batchUpgradeContracts*). However, these functions are fundamentally broken and will always revert due to incorrect access control assumptions and logic errors. The factory deploys MiniSafe proxies and immediately initializes them with a *TimelockController* as the owner.

```
// In deployUpgradeableMiniSafe
addresses.miniSafe = _deployMiniSafe(..., addresses.timelock);
// In MiniSafeAaveUpgradeable.initialize
__Ownable_init(_initialOwner); // _initialOwner is the Timelock
```

The UUPS upgrade mechanism (*upgradeTo*) is protected by onlyOwner. When the factory calls *contractAddress.upgradeTo(...)*, the msg.sender is the Factory, not the Timelock. Consequently, the proxy rejects the call.
The *upgradeSpecificContract* function attempts to verify the target contract using *isMiniSafeContract*.

```
function isMiniSafeContract(address contractAddress) external
view returns (bool) {
 if (contractAddress == miniSafeImplementation || ...) { return
true; }
 return false;
}
```

This function compares the Proxy address (passed as input) with the stored Implementation addresses. Since a proxy address is never equal to its implementation address, this check always returns false, causing the transaction to revert even before the ownership check.

Impact

The "*Emergency Upgrade*" functionality exposed by the factory is completely non-functional. Protocol administrators relying on this mechanism to fix bugs in deployed contracts will find themselves unable to execute upgrades, potentially leaving funds at risk during an actual emergency.

Recommendation

Remove the *upgradeSpecificContract*, *batchUpgradeContracts*, and associated helper functions from the factory.

HIGH SEVERITY ISSUE

[H-02] Malicious Upgrade via Insufficient Timelock Delay

Summary

The *MiniSafeFactoryUpgradeable* contract allows the deployment of *MiniSafe* instances with a *TimelockController* configuration that is insecure. Specifically, the *_validateConfig* function and other deployment helper functions allow a minDelay as short as 1 minute.

While the factory enforces the use of a *Timelock*, a 1-minute delay provides no meaningful security window for users. A malicious actor can:

- Deploy a *MiniSafe* instance using *deployWithRecommendedMultiSig* (or any of the deployment functions), setting themselves as the sole proposer/executor and setting minDelay to 1 minute.
- Wait for users to deposit funds into what appears to be a valid, factory-deployed contract.
- Propose a malicious upgrade (e.g., to an implementation that allows draining funds) via the *Timelock*. Wait 1 minute.
- Execute the upgrade and drain the funds.

Because the delay is so short, users—who are restricted by the *MiniSafe* withdrawal windows (days 28–30)—have no time to react. Even the *breakTimelock* function (which allows emergency exit with a penalty) is ineffective because the attack can be executed faster than human reaction time.

Impact

Users relying on the factory's reputation or the existence of a "Timelock" are susceptible to a complete loss of funds via a "Rug Pull" upgrade. The factory fails to enforce parameters that align with the protocol's security model.

Recommended Mitigation

1. The factory must enforce a minimum delay that provides a sufficient reaction window for users. A minimum of 2 days is recommended to align with the ***EMERGENCY_TIMELOCK*** constant defined in the MiniSafeAaveUpgradeable contract. This ensures that if a malicious upgrade is proposed, users have 48 hours to notice and exit the protocol using breakTimelock.

```
function _validateConfig(UpgradeableConfig memory config) internal pure {
    if (config.proposers.length == 0) revert();
    if (!(config.minDelay >= 2 days && config.minDelay <= 14 days)) revert();
    // Validate proposer addressesfor (uint256 i = 0; i < config.proposers.length;
i++) {
        if (config.proposers[i] == address(0)) revert();
    }
    // Validate delay configuration
if (!(minDelay >= 2 days && minDelay <= 14 days)) revert();

     // Create dynamic arrays from fixed arrayaddress[] memory proposers = new
address[](5);
    address aaveProvider
) external returns (MiniSafeAddresses memory addresses) {
    if (owner == address(0)) revert();
    if (!(minDelay >= 2 days && minDelay <= 14 days)) revert();

    address[] memory proposers = new address[](1);
    address[] memory executors = new address[](1);
}
```

Recommended Mitigation

2. (Optional) Protocol can enforce that the proposer's address must be a single multisig address which has a minimum of 5 signers so that any call made to the timelock contract via the multisig (which is the proposer and canceller) would be signed by atleast 4 signers to initiate an upgrade or cancel an upgrade.

Status: **Resolved**

HIGH SEVERITY ISSUE

[H-3] Depositors Receive Zero Interest

The *MiniSafe* protocol integrates with Aave V3 to generate yield on user deposits. However, due to a fundamental flaw in the share-based accounting system, users receive zero interest on their deposits. The protocol records shares equal to the exact deposit amount (1:1 ratio) and never updates this ratio as interest accrues. When users withdraw, they can only withdraw their original deposit amount, while all accumulated interest remains permanently trapped in the protocol with no mechanism for distribution.
This completely defeats the core value proposition of the protocol - earning yield on savings through Aave integration.

Impact
Depositors get zero interest.

Recommendation
Add Exchange Rate Tracking to *TokenStorage*, or depositors should receive *aToken* after deposits

HIGH SEVERITY ISSUE

[H-4] leaveGroup Always Reverts Due to Missing Balance Credit

leaveGroup() attempts to process a refund by deducting from the user's recorded balance:
updateUserBalance(msg.sender, tokenAddress, refundAmount, false);

However, during normal contributions, the contract never credits *updateUserBalance*() with the contributed amount.
Meaning:

- user deposits tokens into the contract
- but their internal balance remains zero
- when leaving the group, *updateUserBalance*() attempts to deduct *refundAmount* from a non-existent balance

This causes the function to always revert. So one of two failure modes occurs:
Case → *refundAmount* == 0
Effect → user receives no refund despite contributions

Case → refundAmount > 0
Effect → updateUserBalance() underflows or fails
Either outcome prevents successful exit.
Users who contributed:

- cannot leave the group
- cannot recover funds
- are permanently locked in the contract

This is effectively a denial-of-service on exiting members.

Impact

- Users cannot leave thrift groups
- Their funds remain inaccessible
- Group cannot scale down safely

Recommendations

Ensure user balances are credited when contributing.

HIGH SEVERITY ISSUE

[H-5] leaveGroup() emits refund event but never transfers tokens — user funds remain locked in contract

Description

When a user leaves the group, the contract emits a *RefundIssued* event, but no refund is ever transferred to the user.

Instead of sending tokens back, the function calls:

updateUserBalance(msg.sender, tokenAddress, refundAmount, false);

This function only adjusts an internal bookkeeping balance and does not perform any ERC20 transfer.

Therefore:

The protocol signals a refund was issued but the funds remain locked inside the contract

This is especially critical because _removeMemberFromGroup() clears member state before any refund occurs — leaving no recovery path.

If the group becomes inactive due to member removal:

group.isActive = false;

remaining users:

- cannot withdraw
- cannot leave
- cannot trigger refunds

Their contributions are permanently stuck.

Impact

- Users lose access to contributions when leaving group
- Contract emits misleading RefundIssued event
- No mechanism exists to recover locked funds
- Inactive groups trap remaining capital.

Recommendations

Replace ledger adjustment with an actual refund transfer:

IERC20(tokenAddress).safeTransfer(msg.sender, refundAmount);

MEDIUM SEVERITY ISSUE

[M-01] Excess Contribution Amounts Are Permanently Trapped in Contract

In the MiniSafe thrift group functionality, when users make contributions via the makeContribution function, they can send any amount greater than or equal to the required contributionAmount. The function only validates that the amount meets the minimum requirement but does not enforce an exact match or refund excess amounts.

Any tokens sent above the required contribution amount are permanently trapped in the contract with no mechanism for retrieval. The function transfers the full user-specified amount from the user's wallet but only tracks the standard *contributionAmount* in the group's accounting. When payouts are processed, they are calculated based on *contributionAmount* × *memberCount*, meaning the excess tokens are never distributed and remain stuck in the contract forever.

Impact

Excess contributions are permanently locked.

Recommendation

Enforce Exact Contribution Amount.

MEDIUM SEVERITY ISSUE

[M-2] Emergency Withdrawal Timelock Defeats Purpose of Emergency Response Mechanism

Description:

The *executeEmergencyWithdrawal* function in *MiniSafeAaveUpgradeable* implements a 2-day *timelock* before emergency withdrawals can be executed. While timelocks are generally good security practice for administrative functions, applying a timelock to an emergency function fundamentally contradicts its purpose.

In genuine emergency situations, such as protocol exploits, critical vulnerabilities, or external DeFi protocol failures, a 2-day delay renders the emergency mechanism completely ineffective. By the time the timelock expires, the emergency that necessitated the withdrawal may have already resulted in complete loss of funds.

Impact

Emergency Scenarios Where 2-Day Delay Is Fatal.

Recommendation

Remove Timelock

MEDIUM SEVERITY ISSUE

[M-3] Thrift Group Contributions Held in Contract Instead of Earning Yield in Aave

The MiniSafe protocol integrates with Aave V3 to generate yield on user deposits through the personal savings functionality. However, when users make contributions to thrift groups via the makeContribution function, these funds are transferred directly to the MiniSafeAaveUpgradeable contract and held there idle instead of being deposited to Aave to earn yield.

This creates an architectural inconsistency where:

- Personal savings deposits → Sent to Aave → Earn yield
- Thrift group contributions → Held in contract → Earn 0% yield

Given that thrift groups operate on 30-day cycles and funds may sit in the contract for extended periods before payout, this represents a significant missed yield opportunity and contradicts the protocol's core value proposition of earning yield through Aave integration.

Impact

Contributions do not earn yields.

Recommendation

Deposit Contributions to Aave.

Status: Resolved

MEDIUM SEVERITY ISSUE

[M-4] Factory Contract Fails to Track Deployed Proxies isMiniSafeContract Returns False for All Deployed Contracts

Description

The *MiniSafeFactoryUpgradeable* contract contains a critical logic flaw in its isMiniSafeContract function. This function is designed to verify whether a given address is a legitimate MiniSafe contract deployed by the factory. However, it incorrectly checks if the address matches the implementation contract addresses rather than the deployed proxy addresses.

When the factory deploys the MiniSafe system, it creates *ERC1967* proxy contracts that point to the implementations. Users and administrators interact with these proxy addresses, not the implementations. However:

- The factory never stores the deployed proxy addresses
- *isMiniSafeContract* only checks against implementation addresses
- All deployed proxies return false when verified
- Functions relying on this check (upgradeSpecificContract, batchUpgradeContracts) are completely broken

This means the factory's upgrade functionality is non-operational, and any external systems relying on this verification will incorrectly reject legitimate MiniSafe contracts.

Impact

isMiniSafeContract always returns false *upgradeSpecificContract* always reverts *getContractImplementation* always returns zero address

Recommendation

Add Proxy Tracking

Status: Resolved

LOW SEVERITY ISSUE

[L-1] updatePoolDataProvider and updateAavePool Emit Identical Event - Impossible to Distinguish Configuration Changes

Description

In the MiniSafeAaveIntegrationUpgradeable contract, two distinct administrative functions—updatePoolDataProvider and updateAavePool—emit the same AavePoolUpdated event. This makes it impossible for off-chain systems, block explorers, indexers, and monitoring tools to distinguish between updates to the Pool Data Provider versus the Aave Pool contract.

Both contracts serve fundamentally different purposes in the Aave V3 architecture:

- IPool: Handles state-changing operations (supply, withdraw, borrow, repay)
- IPoolDataProvider: Handles view-only queries (getReserveTokensAddresses, getUserReserveData)

When an administrator updates either contract, the emitted event is identical, creating ambiguity in audit trails, monitoring systems, and incident response procedures.

Impact

Cannot determine which contract was updated.

Recommendation

Add Separate Event for PoolDataProvider

Status: Resolved

LOW SEVERITY ISSUE

[L-2] Deposit Timestamp Is Overwritten and Never Used

Description

Each new deposit overwrites the existing deposit timestamp, and the stored deposit time is never referenced or enforced anywhere in the protocol logic. This results in a state variable that:

- Does not represent the original deposit time
- Is not used for validation, accounting, lockups, cooldowns, or rewards
- Can be arbitrarily refreshed by making a new deposit

Impact

Incorrect State Representation

Recommendation

Enforce Deposit Time Properly

LOW SEVERITY ISSUE

[L-3] Misleading Error Message in Withdrawal Logic

require(amountWithdrawn > 0, "aToken address not found");

This message is incorrect. The *aToken* address is already verified earlier. This failure indicates a zero withdrawal, not a missing address.

Recommendation

Update the revert message to accurately reflect the failure condition.