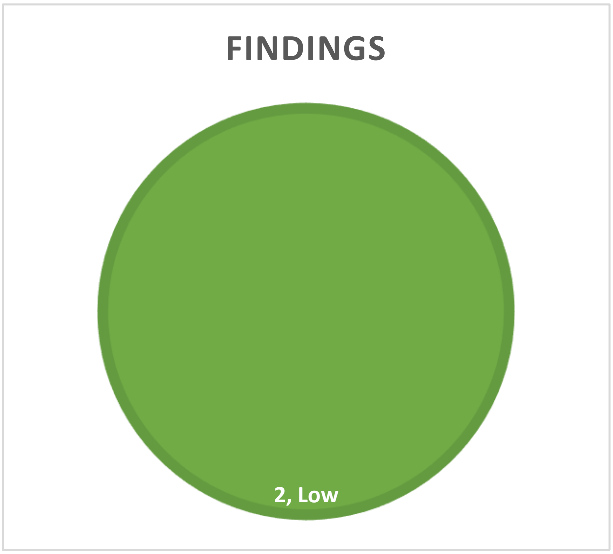
Trust Security

Smart Contract Audit

Merit Systems Escrow Contract

01/05/2025

Executive summary

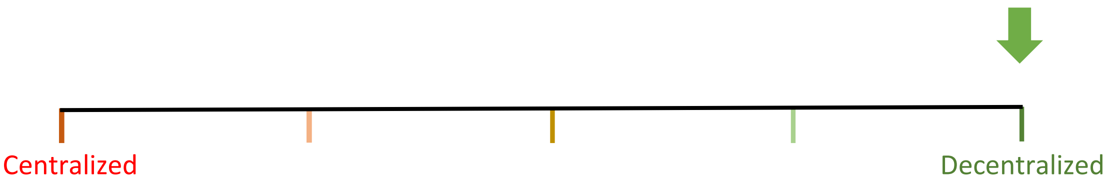


Category	Escrow
Audited file count	2
Lines of Code	250
Auditor	Trust
Time period	03/04/25-05/04/25

Findings

Severity	Total	Fixed	Acknowledged
High	0	-	-
Medium	0	-	-
Low	2	-	-

Centralization score



Signature

Digitally signed by Trust
Date: 2025.05.01 12:59:39 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	05/04/2025	Client report
0.2	06/04/2025	Mitigation review
0.3	01/05/2025	Additional commit review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- src/Escrow.sol
- scripts/Deploy.Base.s.sol

Repository details

- **Repository URL:** https://github.com/Merit-Systems/ledger
- **Commit hash:** 64a8cc8a9db4231992f3412418577ca9ae271725
- **Mitigation review commit hashes:**
    - a945e81ddac990cdebd4f55aa3483944d6d08d24
    - 6b733c855177be7dc2ffab98ad8ff6dc1317a12b
- **Additional changes commit hash**: b836e778240c383b22f99d677cfd39d15d7c4aa0

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	**Rating**	**Comments**
Code complexity	**Excellent**	Project kept code as simple as possible, reducing attack risks
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Excellent**	Project does not introduce significant centralization risks.

Findings

Low severity findings

TRST-L-1 Batch functions may randomly fail or be griefed by bots

- **Category:** DOS issues
- **Source:** Escrow.sol
- **Status:** Acknowledged

Description

The contract offers a batch wrapper on *claim()* and *reclaim()*. Note that a call to any of these functions for the same deposit ID will fail due it being already claimed. Therefore, the batch action is sensitive to any of the sub-actions being executed between the time the batch is constructed, to the time it is executed. It could happen by coincidence, or through a targeted attack by viewing the mempool. On Base, the mempool is private, so the risk is reduced.

Recommended mitigation

There are two options:

- Refactor the original *claim()*, *reclaim()* functions so that they exit without revert in case the claim / reclaim was executed.
- Wrap the batch functions with a try/catch, and make the calls to *claim()*, *reclaim()* external calls.

Team response

Acknowledged.

TRST-L-2 Race condition on the ownership of funds after the claim deadline could lead to disputes

- **Category:** Time-sensitivity issues
- **Source:** Escrow.sol
- **Status:** Acknowledged

Description

The code allows depositors to reclaim the deposit after the **claimDeadline**, while recipients can claim the deposit at *any* time. This creates a dispute opportunity when the user reclaims, but is frontrun by a recipient, or in fact anyone calling *claim()* on their behalf. At any moment, it should be clear who effectively has ownership of the funds, so as to not have any grounds for escalation.

Also, note the *reclaim()* function reverts with **STILL_CLAIMABLE** before the deadline, which lines up with the desired behavior – recipient should not be able to claim past the deadline.

Recommended mitigation

Verify in *claim()* that the **claimDeadline** has not passed.

Team response

The team does not intend to handle this case in a specific way. Whoever executes this first after the **claimDeadline** gets the tokens.

Additional recommendations

TRST-R-1 Reduce spam risks for depositors and recipients

In the current implementation, anyone can deposit on behalf of another user, and specify any recipient. The minimum deposit amount is 1 wei. That means it is easy and cheap to make many fake deposits which the user is not interested in looking at. This could be handled in the frontend, but it should be mentioned that the view functions will not necessarily be usable for clear information.

TRST-R-2 Define upper bound for the claim period

Currently the **claimDeadline** could be up to **MAX_UINT256**. In case user makes a mistake and passes a large amount, it would be better to prevent it by enforcing a reasonable maximum.

TRST-R-3 Rounding against the user to avoid cheating the fees

The fee amount is calculated in *deposit()* using *mulDivDown()*, rounding down the fee. The safer behavior would be to round it up, so that user does not craft specific small numbers to avoid the fee altogether (that would be profitable when gas is very cheap).

TRST-R-4 Improve validations

Some additional checks could be added:

- In *setCanClaim()*, the code returns if the target status is the current status. However, there is only a valid use case for True -> True state transition. Consider reverting in the False -> False scenario to better fortify the code.
- When adding or removing whitelisted tokens, there are checks that the operation changes the state. However, in the constructor there is no *require()* statement. Consider adding that to ensure sane deployment parameters.

TRST-R-5 Consider gas limits when setting batchDepositLimit

The owner can set the number of deposits in a batch through the **batchDepositLimit** variable. While it is currently set to 500 by the deployment scripts, it should be checked if such a large loop with multiple token transfers and storage updates is possible. Consider running tests to view the gas spending and make sure it is below the block gas limit. There are no security implications, but it may create an issue where the frontend or other clients batch too many deposits together and make all the TXs revert.

TRST-R-6 Avoid unusual usage of Create2 during deployment

The deployment script makes use of OpenZeppelin Create2 in order to deploy the Escrow contract.

```
vm.startBroadcast();
address deployed = Create2.deploy(0, Params.SALT, bytecode);
vm.stopBroadcast();
```

That is quite unusual, as the simulation is ran through an EOA sender account which can't actually execute the CREATE2 instruction. It is unclear if the output broadcast of such a script is reliable.

It is recommended to use the standard method provided by Foundry book, which deploys the contract uses the Deterministic Deploy address. It's important to follow all the steps in the manual for correct operation.

Centralization risks

TRST-CR-1 Owner can set fees up to 10%

The contract admin is able to set fees at any time, up to a maximum of 10%. In the worst case, this could be misused to surprise users depositing under assumption of a lower fee. The threat could be mitigated with a slippage parameter for taken fees, or accepted as a known, reasonable risk.

TRST-CR-2 Valid signatures for claiming can be revoked by the owner

The *setSigner()* function allows the owner to update the **signer** parameter used to verify a signature sent to the *setCanClaim()* function. This means that a signature currently considered valid can be invalidated, and user may lose the right to *set* the **canClaim** to true. Note that user can always make use of the signature immediately and call *claim()*, to avoid trusting the owner.