Blaize.Security

Midcontract

MIDCONTRACT
SMART CONTRACT AUDIT

January 20th, 2025 / V. 1.0

Table of Contents

Executive Summary

During the audit, we examined the security of the smart contracts for the Midcontract protocol. Our task was to identify and describe any security issues within the platform's smart contracts. This report presents the findings of the security audit of the **Midcontract** smart contracts conducted between **December 23rd** and **January 20th**.

Blaize.Security conducted an in-depth audit of the Midcontract smart contracts. The smart contracts represent a platform for clients and contractors where users can regulate their business relationships. It allows users to create their own escrows and submit tasks by contractors.

The security team decomposed the protocol, verified the integrity of the business logic, funds flow, access control systems, and user workflows. The team has ensures the correctness of creation, submitting and canceling of the contracts for hourly, fixed price and milestone work model.

The security team currently evaluates the project as **Secure**. The Midcontract resolved all issues, and the security team notes good approach in contract's deployment and maintenance of all the flows.

Auditing strategy and Techniques applied/Procedure

Blaize.Security auditors start the audit by developing an auditing strategy - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

MANUAL AUDIT STAGE

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

and a wide spectrum of other vulnerable areas.

FOR ADVANCED COMPONENTS:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

TESTING STAGE:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

POST-AUDIT STEPS RECOMMENDED

To ensure the security of the contract, the **Blaize.Security** team suggests that the team follow post-audit steps:

1. Request audits of other protocol components (dApp, backend, wallet, blockchain, etc) from Blaize Security
2. Request consulting and deployment overwatch services provided by Blaize Security
3. Launch active protection over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
4. Launch a **bug bounty program** to encourage further active analysis of the smart contracts.
5. Request post-deployment assessment service provided by Blaize Security to ensure the correctness of the configuration, settings, cross-connections of deployed entities and live functioning

Audit Rating

Score: **10** /10

	RATING
Security	10
Logic optimization	10
Code quality	10
Testing suite	10
Documentation	10

Security: General mark for the security of the protocol.
The main mark for the audit qualification.

Logic optimization: Evaluation of how optimal the implementation is, including presence of extra/unused code, uncovered/extra cases, gas (or its analog) optimization, memory management optimization, etc
Code quality: Evaluation of best practices followed, code readability, structure and convenience of further development
Testing suite: Availability of the native tests suite, level of logic coverage, checks of critical areas being covered.
Documentation: Availability and quality of the documentation, coverage of core functionality and user flows: whitepaper, gitbook, readme, specs, natspec, comments in the code and other possible forms of documentation.

SECURITY RATING CALCULATION

Approximate weight of unresolved issues.
Critical: -3 points
High: -2 points
Medium: -0.5 points
Low: -0.1 points
Informational: -0.1 point (in general, depends on the context)

Note: additional concerns, violated checklist items (including standard vulnerabilities), and verified backdoors may influence the final mark and weight of certain issues.

Starting with a perfect score of 10:

Critical issues: 2 issue (2 resolved): 0 points deducted

High issues: 0 issue (0 resolved): 0 points deducted

Medium issues: 2 issues (2 resolved): 0 points deducted

Low issues: 2 issues (2 resolved): 0 points deducted

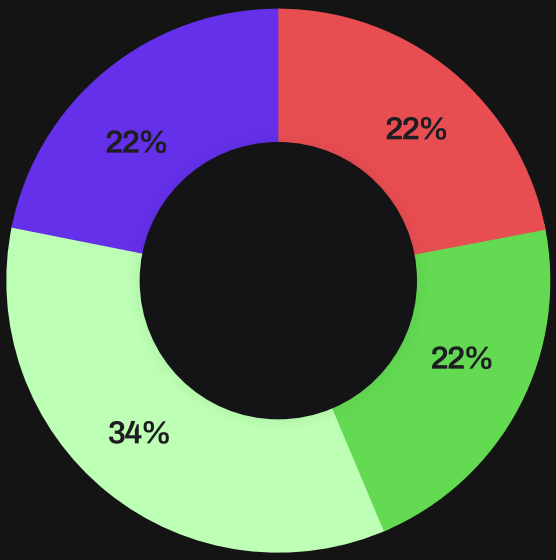
Informational issues: 3 issues (3 resolved): 0 points deducted

Security rating = 10

Technical Summary

THE GRAPH OF VULNERABILITIES DISTRIBUTION:

Critical
High
Medium
Low
Info



The table below shows the number of the detected issues and their severity. A total of 5 problems were found. 5 issues were fixed or verified by the Customer's team.

	FOUND	FIXED/VERIFIED
Critical	2	2
High	0	0
Medium	2	2
Low	2	2
Info	3	3

Best practices and optimizations

- 5 items resolved

Section is marked as **resolved**

SEVERITY DEFINITION

CRITICAL

The system contains several issues ranked as very seriousand dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

HIGH

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

MEDIUM

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

LOW

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

INFO

The issue has no impact on the contract's ability to operate, yet is relevant for best practices. Or this status can be assigned to the issues related to the suspicious activity or substandard business logic decisions which cannot be classified without the comments from the team (and can be re-classified on the later audit stages).

Issues reviewed by the team can get the next statuses:
Resolved: issue is resolved by an appropriate patch or changes in the business logic
Verified: the team provided sufficient evidences that the issue describes desired behavior
Unresolved: neither path nor comments provided by the team, or they are not sufficient to resolve the issue
Acknowledged: the team accepts the misbehavior and connected risks

Audit Scope

Language/Technology: **Solidity**

Blockchain: **Polygon**

The scope of the project includes:

- Enums.sol
- ERC1271.sol
- EscrowAccountRecovery.sol
- EscrowAdminManager.sol
- EscrowFeeManager.sol
- EscrowRegistry.sol
- EscrowFactory.sol
- EscrowFixedPrice.sol
- EscrowHourly.sol
- EscrowMilestone.sol

Repository: https://github.com/midcontract/contracts

The source code of the smart contract was taken from the branch:
main. Fixes occurred in the branch fix/audit.
Initial commit:

- 9f6f8949b959875f8e012d85895a8b431862aee4

Final commit:

- 5733dc3667ae5a3e9db56d38d6331aa84cbf747e

Protocol overview

DESCRIPTION

Midcontract is a protocol which regulates the relationship between clients and contractors. It allows anyone to create their own escrows where contracts can be created. These contracts can be funded by certain ERC-20 tokens. After creation of the contract, the contractors can pick up the contracts and claim funds once the contract is marked as approved. Escrow smart contracts also allows clients to make requests to return funds or resolve disputes between client and contractor.
The protocol consists of service smart contracts which serves several supporting roles such as fee calculation, role-management, etc, factory for escrow creation and 3 implementations of the escrow. Each escrow differs by how funds are allocated for contractors: by fixed price, hourly or each milestone.

EscrowAccountRecovery is a service smart contract which allows the recovery of access to the smart contract for clients or contractors. The recovery can be initiated by specific users with the role Guardian and executed by a new address of client or contractor.

EscrowAdminManager is a service smart contract which manages different roles within the protocol. Utilizes OwnedRole by SolBase library. Allows only the owner of a smart contract to grant or remove roles for users.

EscrowFeeManager is a service smart contract which calculates the fees during deposits or claims in the escrows. The contract utilizes different fees such as default, user-, instance-, contract-specific. Each fee has its own priority. For example, if a fee is specified for a contract, then a contract-specific fee will be used. It is specified for the user, then user-specific-fee will be used. If no fee is specified, the default fees are applied.

EscrowRegistry contains a list of smart contracts' addresses of the protocol including the implementations of escrows. It also aggregates the blacklist and the list of supported payment tokens.

ERC1271 - a utility smart contract which validates the signatures following the ERC1271 standard.

Enums - library which contains different enums used across the protocol.

EscrowFactory is a factory smart contract which allows anyone to create escrows. Users are able to specify the type of the escrow: fixed price, hourly or milestone.

EscrowFixedPrice is an escrow smart contract with a fixed price payment model. It allows the client to create a contract and specify an amount of payment token which the contractor will receive upon fulfillment of the contract.

EscrowHourly is an escrow smart contract with an hourly payment model. It allows client to deposit payment every week and approve a certain amount of funds to be claimable for the contractor. Thus, the contractor can claim funds from every week of the contract during which the work was done.

EscrowMilestone is an escrow smart contract which allows clients to divide scope of contract into separate milestones. Each milestone can have its own amount of payment funds and contractor. Upon approval of work of certain milestone, the contractor can claim the funds.

ROLES AND RESPONSIBILITIES

1. EscrowAccountRecovery

Guardian – initiates recovery of the ownership of the escrow (for clients) or ownership of the contracts (for contractors).
Client/Contractor (User) – cancel recovery (for old accounts) or execute recovery (for new accounts).

2. EscrowAdminManager

Owner – grant or revoke other roles such as Admin, Guardian, Strategist, DAO.

3. EscrowFeeManager

Admin - set different kinds of fee percentage.

4. EscrowRegistry

Owner - set addresses of smart contracts, maintain blacklist of users and list of payment tokens.
Blacklisted user - a user who is restricted from participating in transactions within the protocol to ensure compliance and enhance security.

5. EscrowFactory

Owner - update address of the registry, pause/unpause deployment of escrows.
Users - can deploy escrows for themselves.

6. EscrowFixedPrice/EscrowHourly/EscrowMilestone

Client - create new contracts/weeks/milestones with correct settings, refill funds into contracts, approve contractors' work, initiate refund or dispute, withdraw refunded tokens.
Contractor - submit for the work, claim funds, create disputes.
Admin - resolve disputes, update address of registry.
Owner - update address of admin registry.
Account Recovery SC - transfer ownership of client or contractors.

LIST OF VALUABLE ASSETS

1. EscrowFixedPrice/EscrowHourly/EscrowMilestone

Payment tokens - ERC-20 tokens used as payment for contractors and protocol fees. Filled by the client when contracts are created or refilled. Only supported tokens can be used.

SETTINGS

1. EscrowAccountRecovery

recoveryPeriod - period during which recovery can be canceled. After this period passes, the recovery can be executed.

2. EscrowFeeManager

contractSpecificFees - coverage and claim fees for a specific contract. Have the first priority.
instanceFees - coverage and claim fees for a specific instance of smart contract. Have the second priority.
userSpecificFees - coverage and claim fees for a specific user. Have the third priority.
defaultFees - default coverage and claim fees. Used by default if no other fees are specified.

3. EscrowRegistry

escrowFixedPrice - implementation of Fixed Price Escrow. Used by Factory to deploy new proxy instances.
escrowMilestone - implementation of Milestone Escrow. Used by Factory to deploy new proxy instances.
escrowHourly - implementation of Hourly Escrow. Used by Factory to deploy new proxy instances.
treasury - an address used to receive fees from all the escrows.
paymentTokens - a list of supported payment tokens which can be specified by the clients during creation of the contracts.
blacklist - a blacklist to block actions of certain users within the protocol.

4. EscrowFactory

factoryNonce - contains current nonce of the user necessary for the deployment of escrows.
existingEscrow - a map of existing escrows.

5. EscrowFixedPrice

deposits - contains the list of details per each contract. These details include:

- Address of the contractor
- Payment token
- Amount of payment token deposited in the contract
- Claimable amount for contractor
- Withdrawable amount for client
- Contractor data about the work
- Fee config
- Status of the contract

6. EscrowHourly

contractDetails - contains a list of details per each contract. These details are:

- Address of the contractor
- Payment token
- Prepayment amount
- Withdrawable amount for client
- Fee config
- Status of the contract

weeklyEntries - contains information about every week of the contract thus storing the payment cycle. The details include:

- Claimable amount for contractor
- Status of the week

7. EscrowMilestone

maxMilestones - a maximum number of milestones which can be processed in a single deposit transaction. Doesn't restrict the number of milestones per contract.

contractMilestones - contains a list of milestones per contract. Each milestone contains the following information:

- Address of the contractor
- Amount of the payment token
- Claimable amount for contractor
- Withdrawable amount for client
- Contractor data about the work
- Fee config
- Status of the milestone

milestoneDetails - contains additional details regarding the milestone:

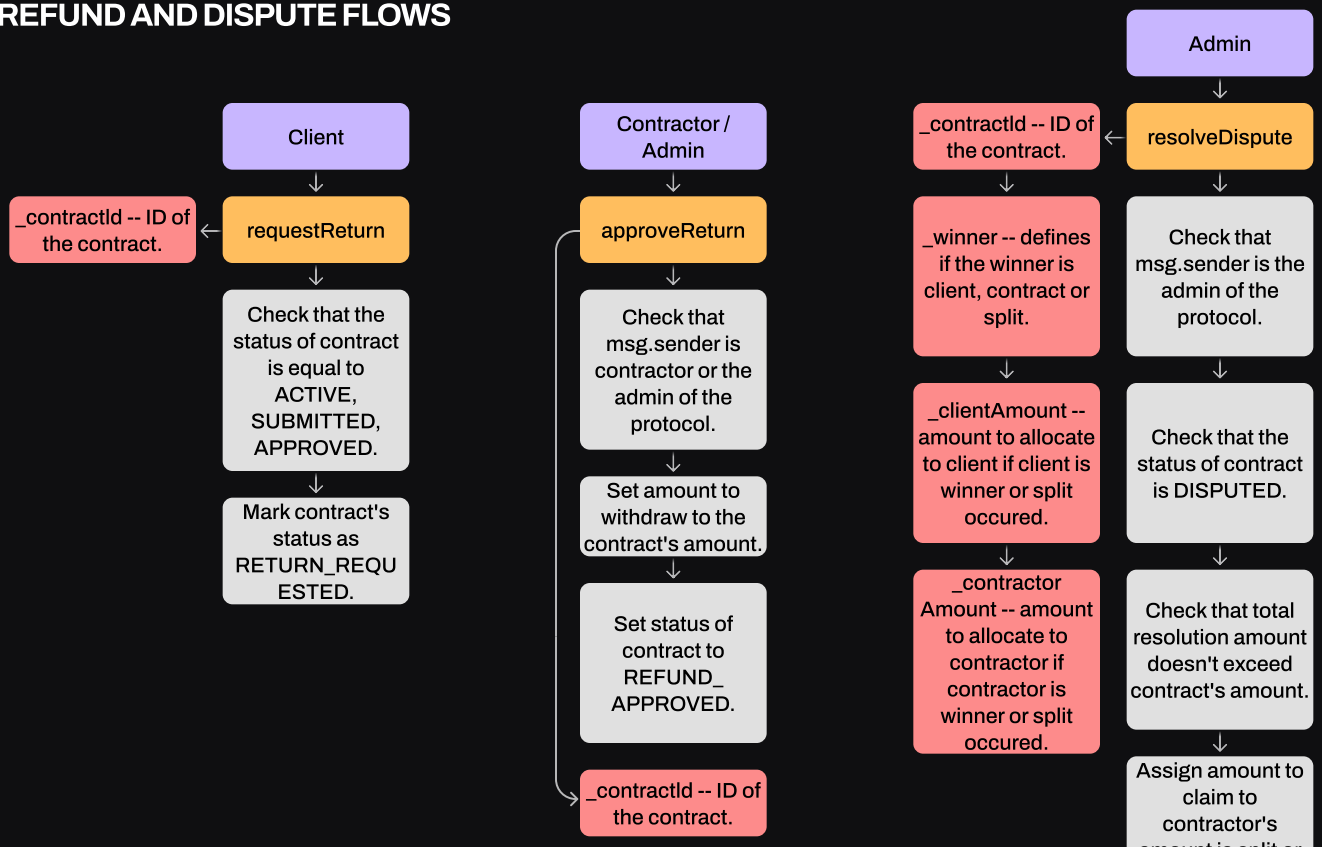
- Address of the payment token
- Deposit amount
- Winner of a dispute if there is any

DEPLOYMENT SCRIPTS

All scripts are located in script\deploy location. Scripts perform deployment of all the module smart contracts, factory and implementation of all the escrows which are then set in the Registry.

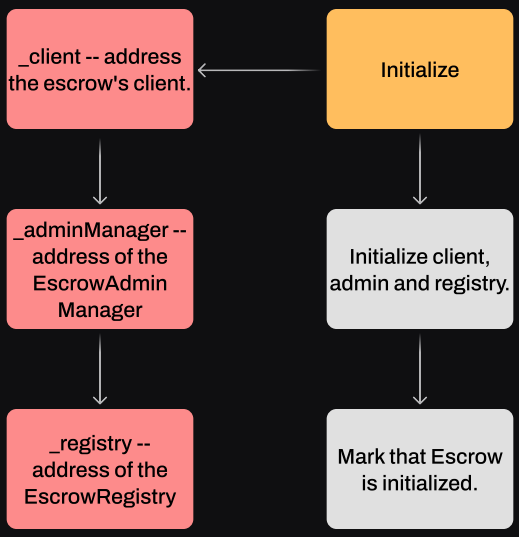
ESCROWFIXEDPRICE

REFUND AND DISPUTE FLOWS



ESCROWFIXEDPRICE

DEPLOYMENT



ROLE-RESTRICTED SETTERS

ESCROWFIXEDPRICE

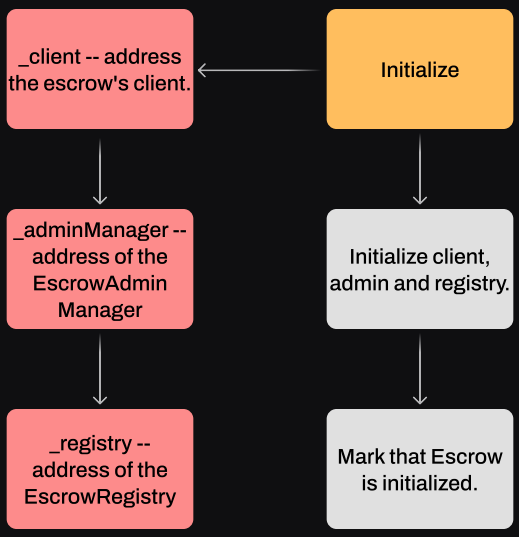
CONTRACT LIFECYCLE

ESCROWFIXEDPRICE

CONTRACT LIFECYCLE

ESCROWHOURLY

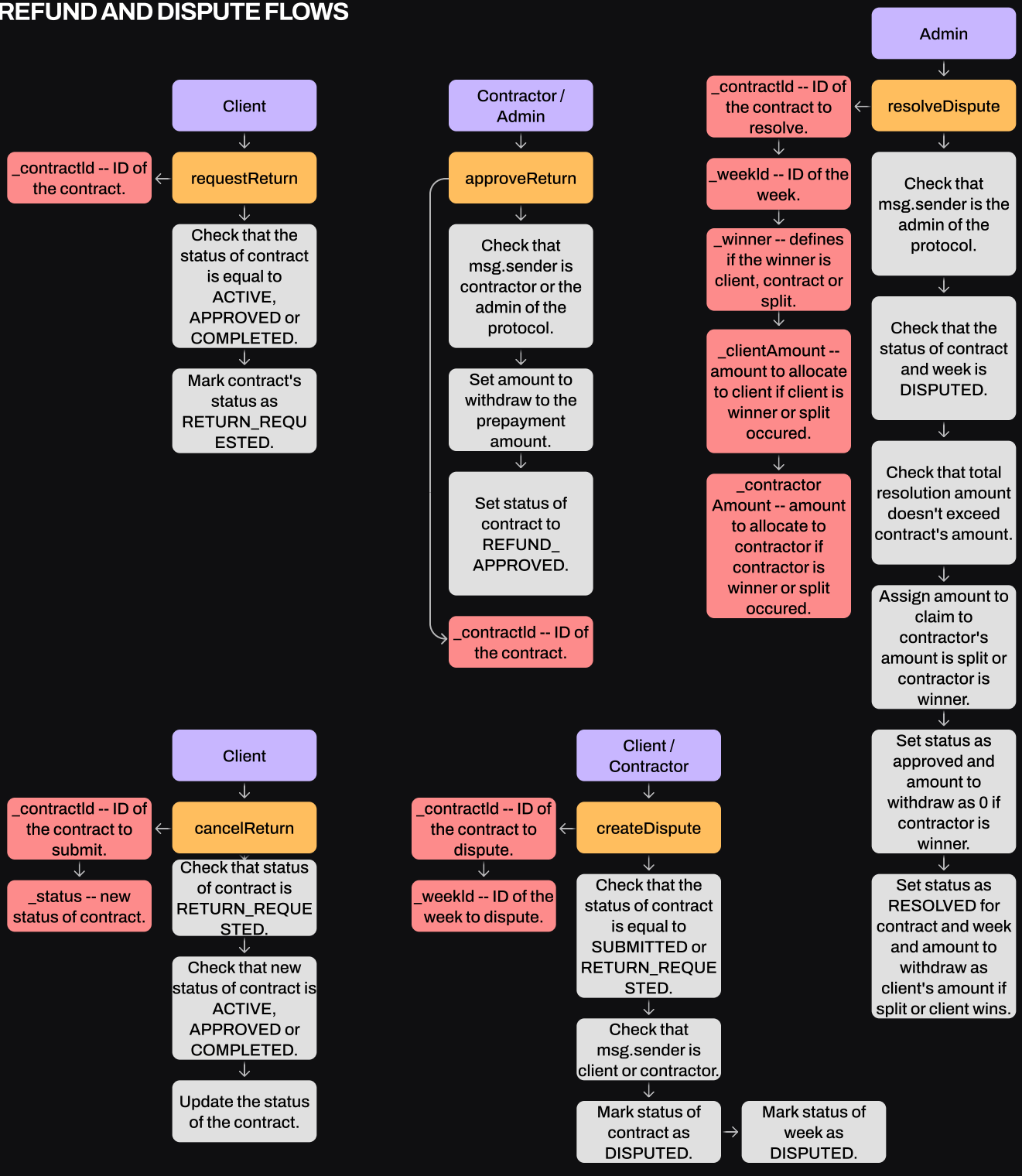
DEPLOYMENT



ROLE-RESTRICTED SETTERS

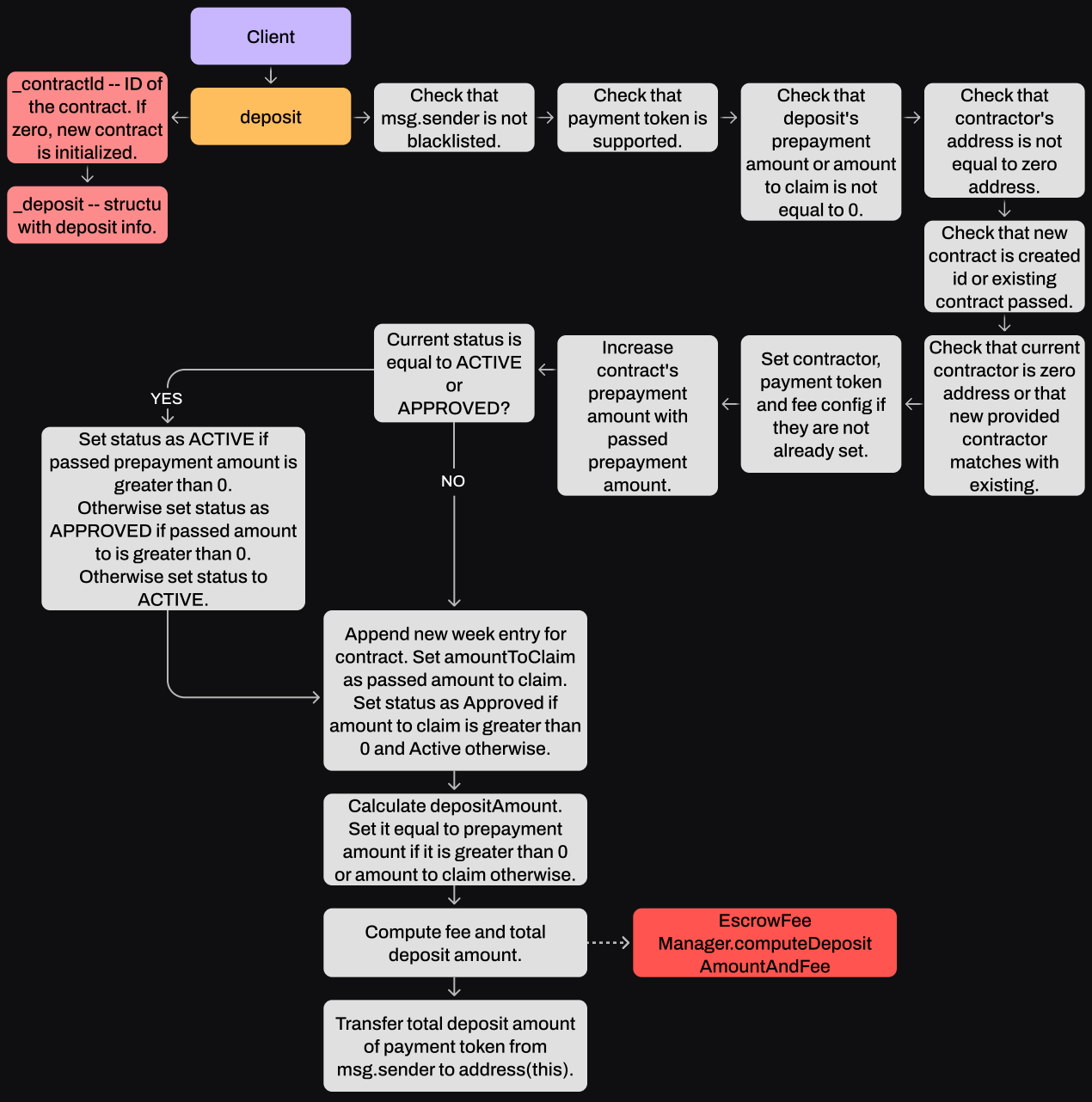
ESCROWHOURLY

REFUND AND DISPUTE FLOWS



ESCROWHOURLY

CONTRACT LIFECYCLE

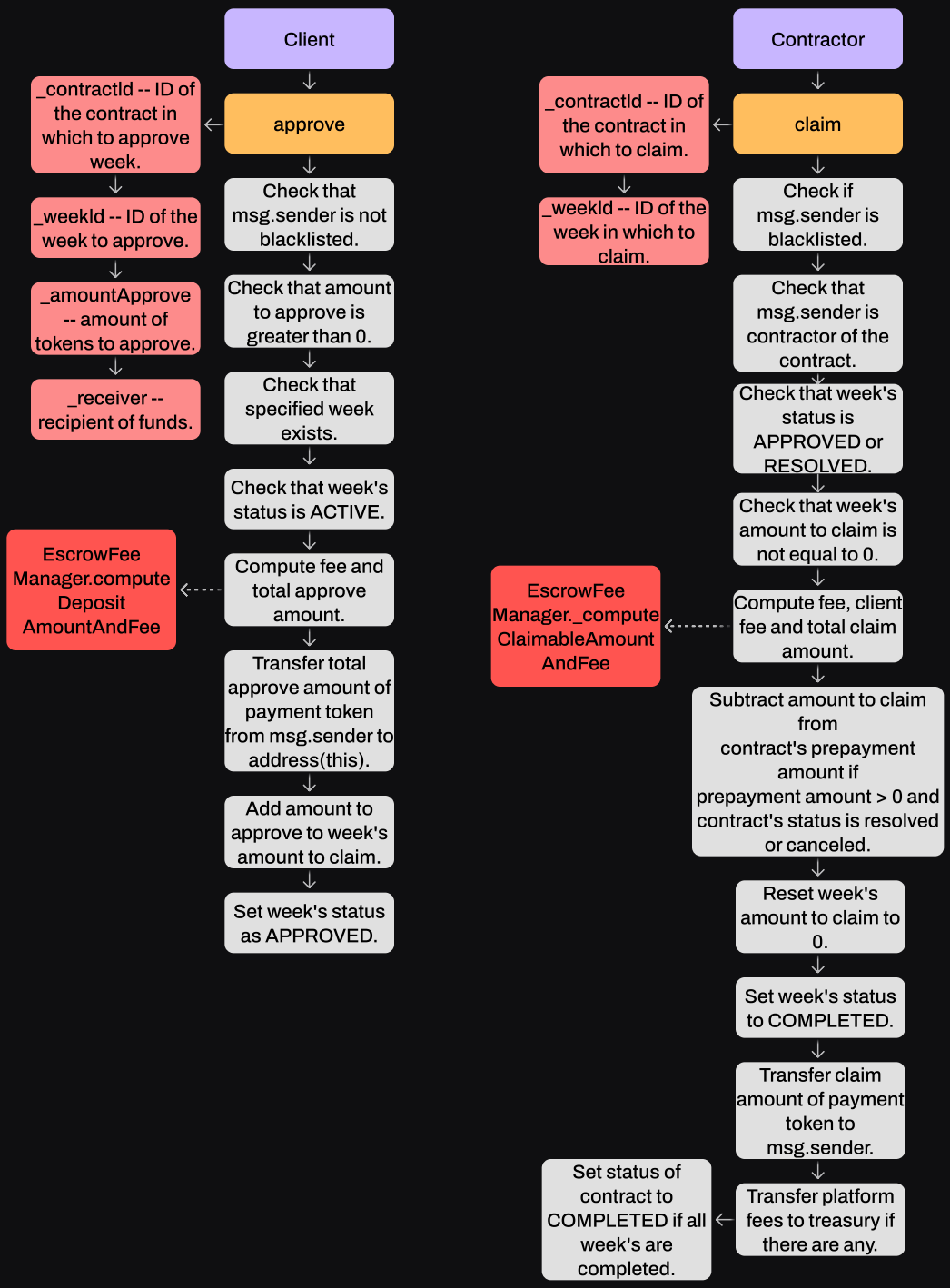


ESCROWHOURLY

CONTRACT LIFECYCLE

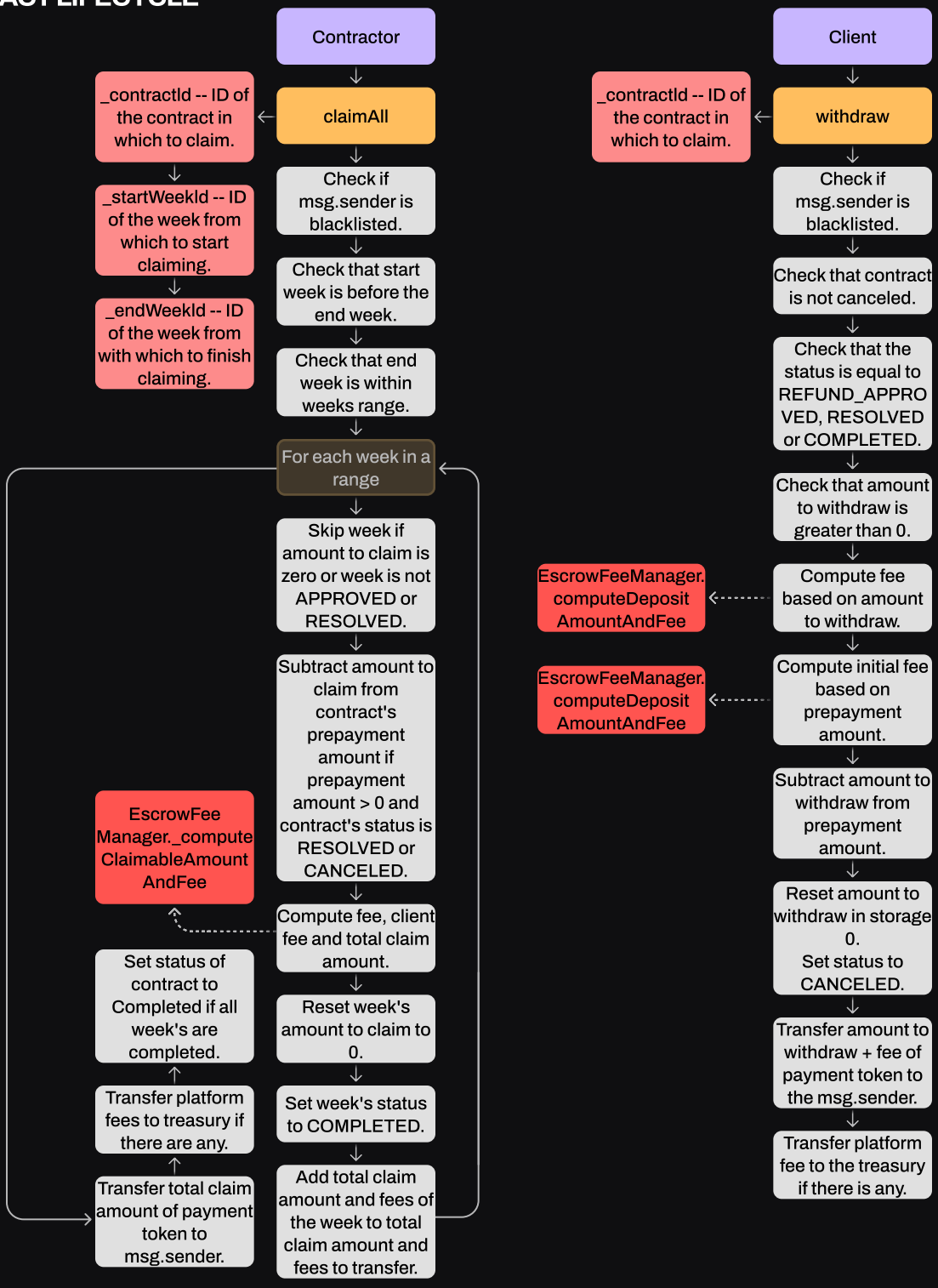
ESCROWHOURLY

CONTRACT LIFECYCLE



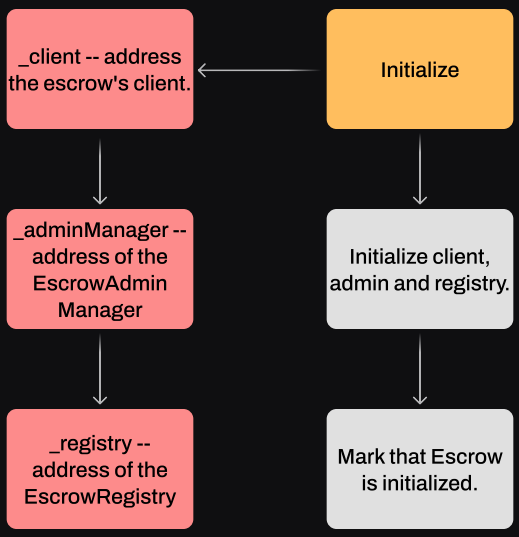
ESCROWHOURLY

CONTRACT LIFECYCLE

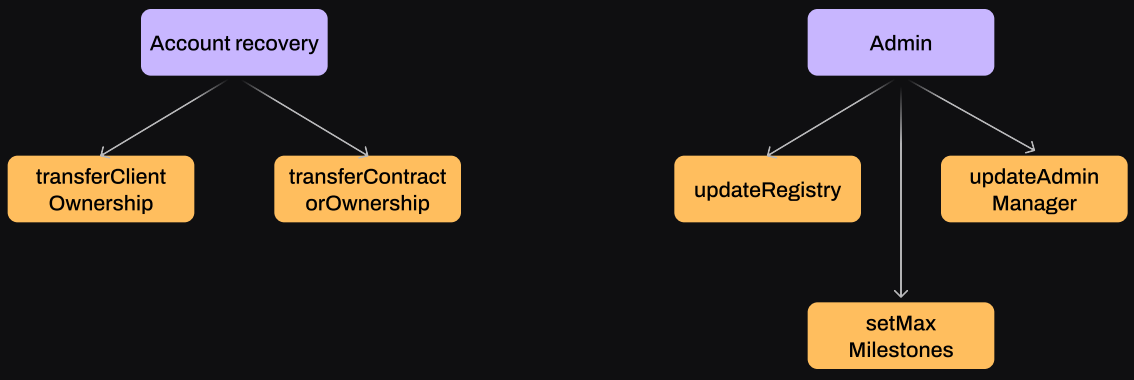


ESCROWMILESTONE

DEPLOYMENT

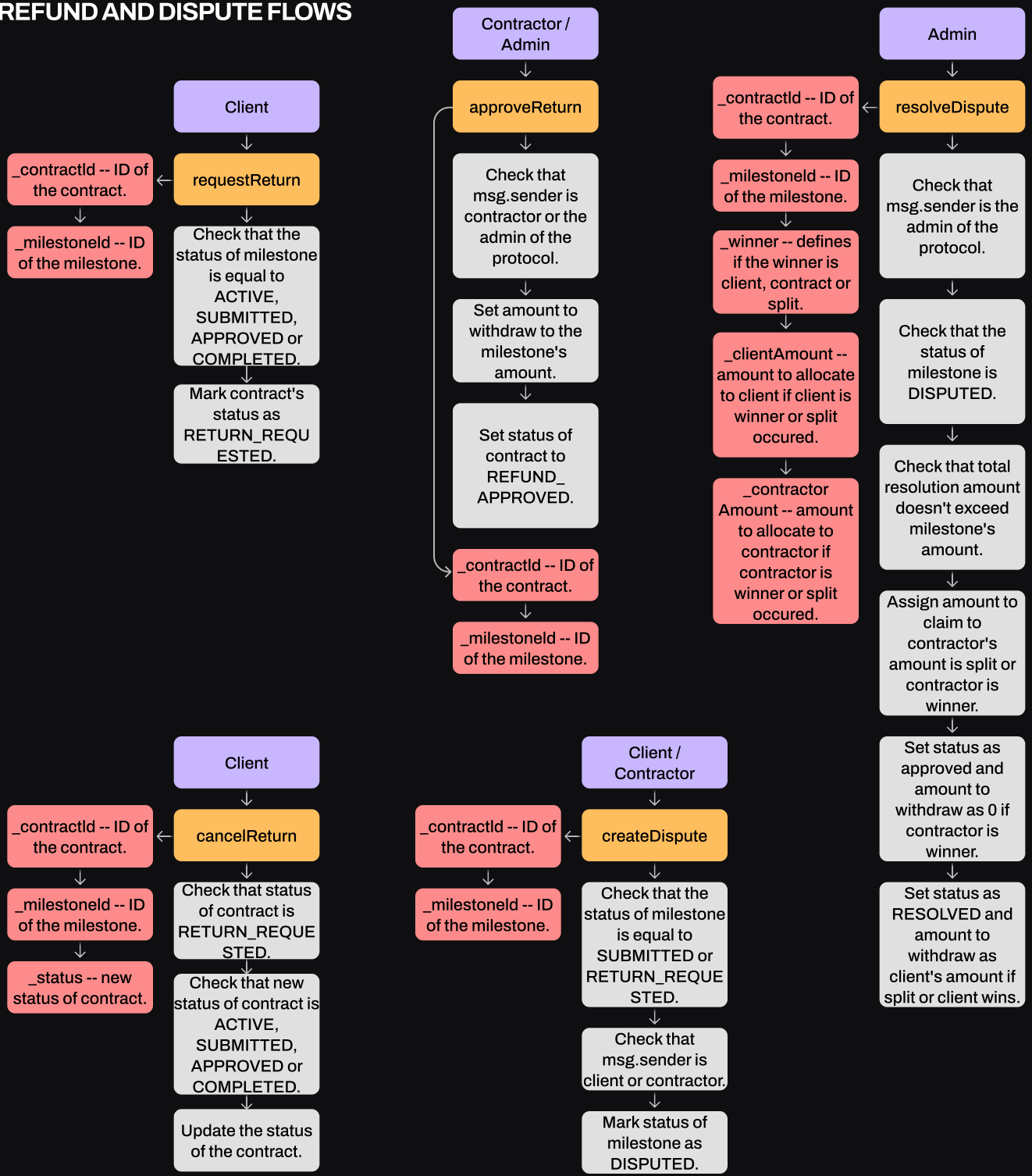


ROLE-RESTRICTED SETTERS



ESCROWMILESTONE

REFUND AND DISPUTE FLOWS



ESCROWMILESTONE

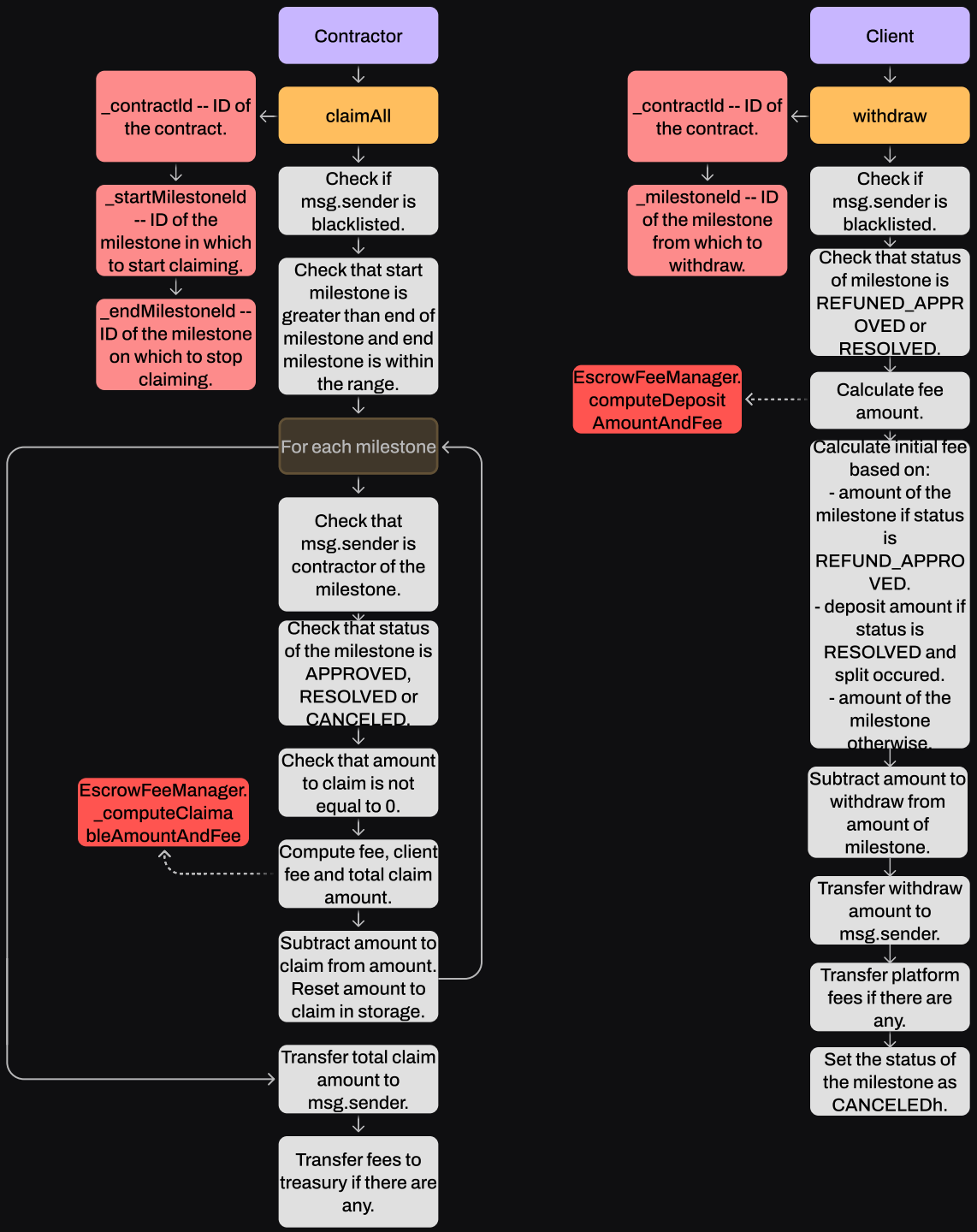
CONTRACT LIFECYCLE

ESCROWMILESTONE

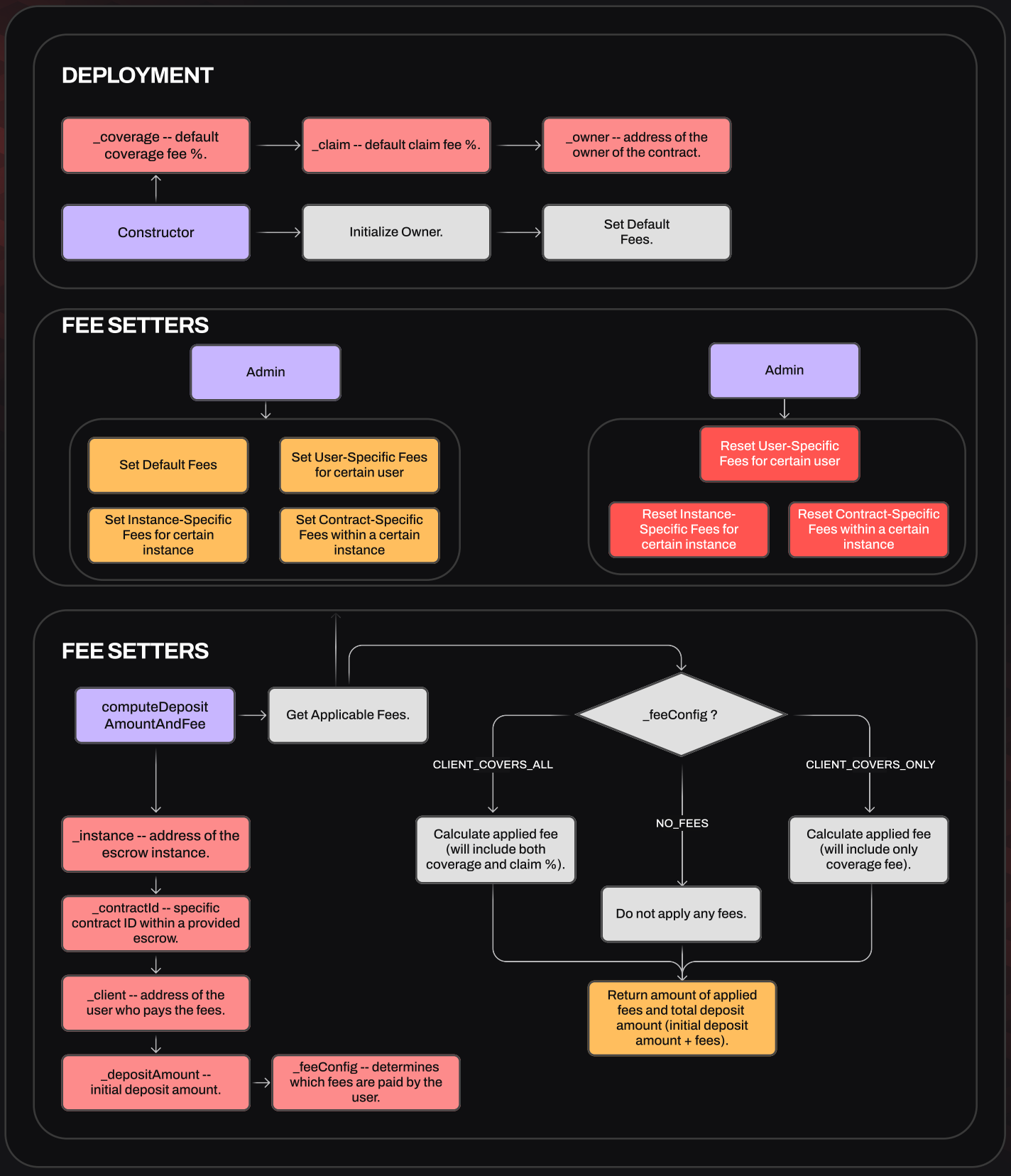
CONTRACT LIFECYCLE

ESCROWMILESTONE

CONTRACT LIFECYCLE

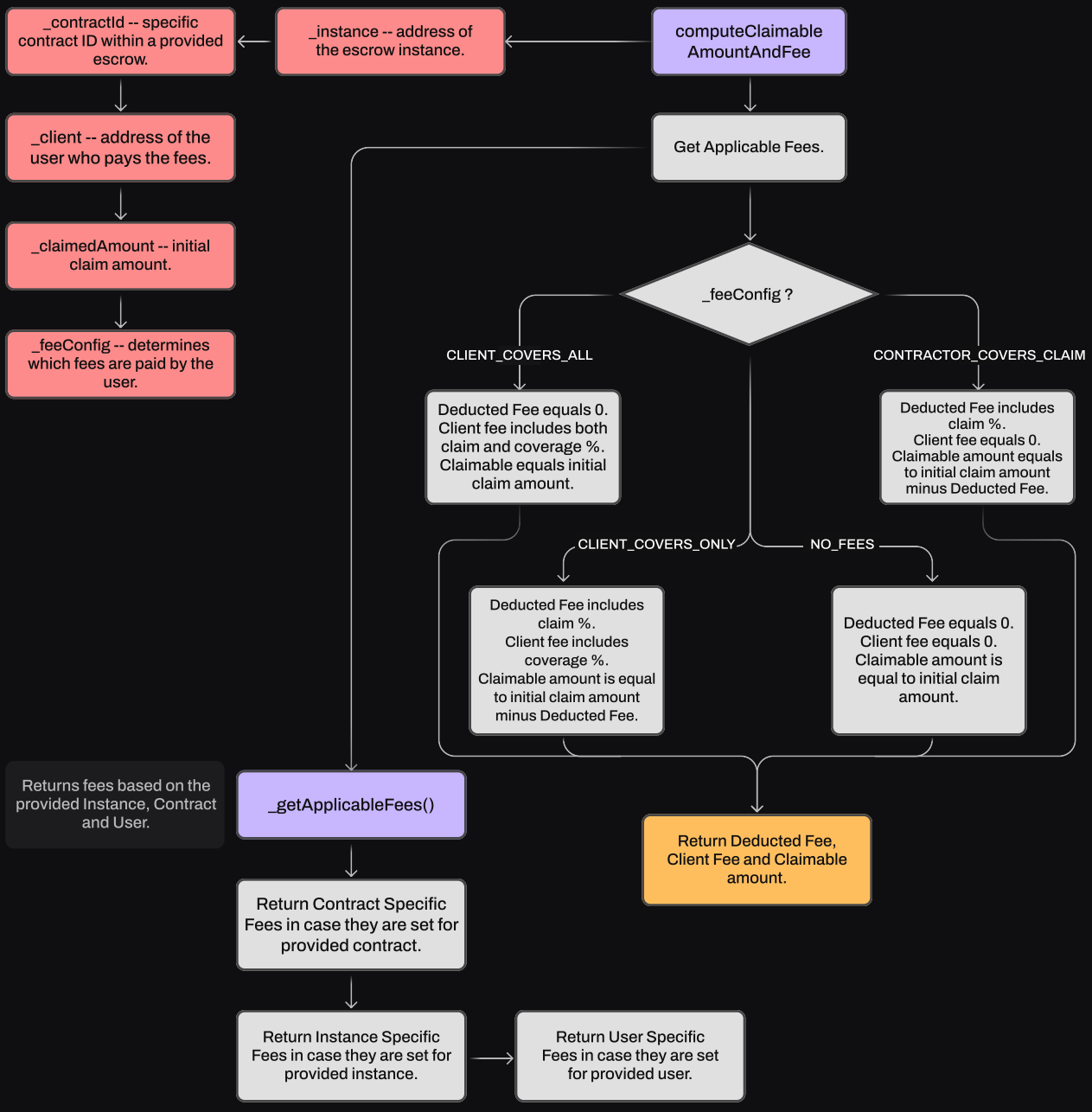


ESCROWFEEMANAGER



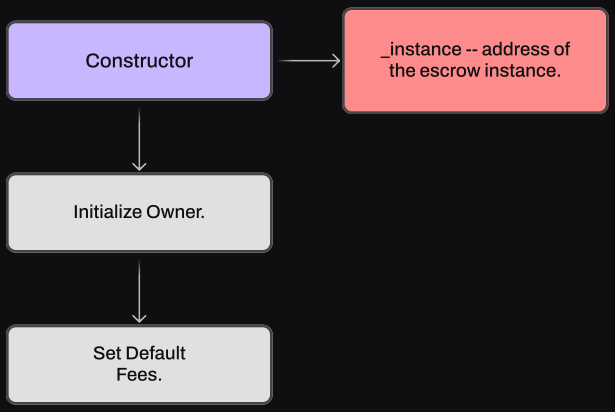
ESCROWFEEMANAGER

FEE SETTERS

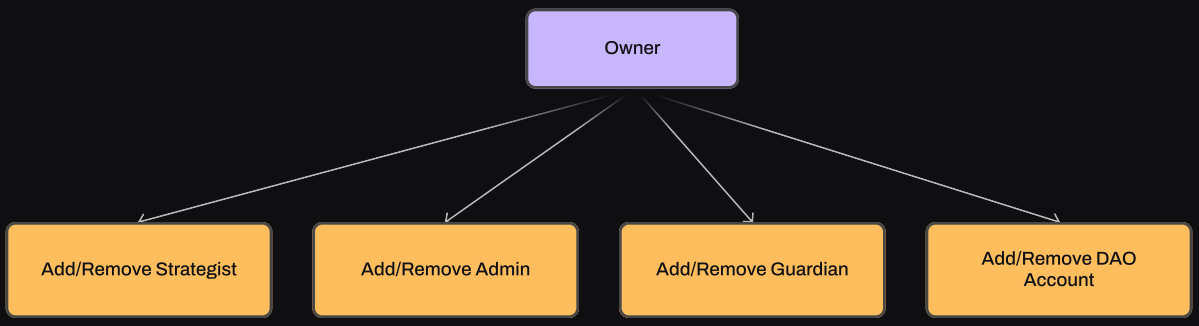


ESCROWADMINMANAGER

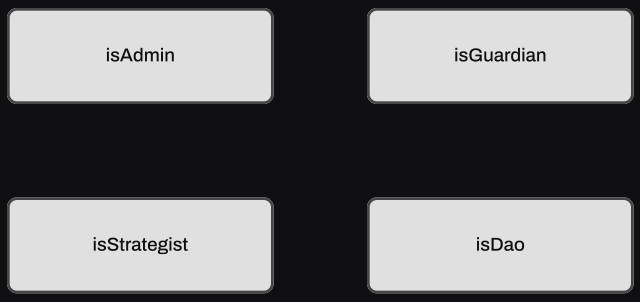
DEPLOYMENT



OWNER CONTROLLED SETTERS

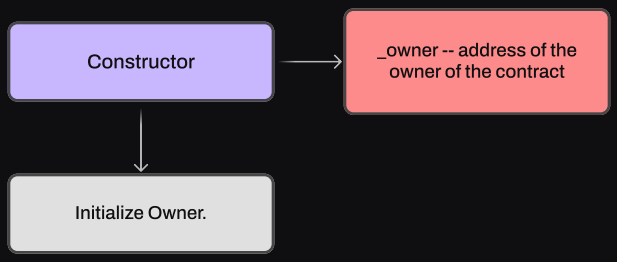


PUBLIC GETTERS FOR ROLE VALIDATION

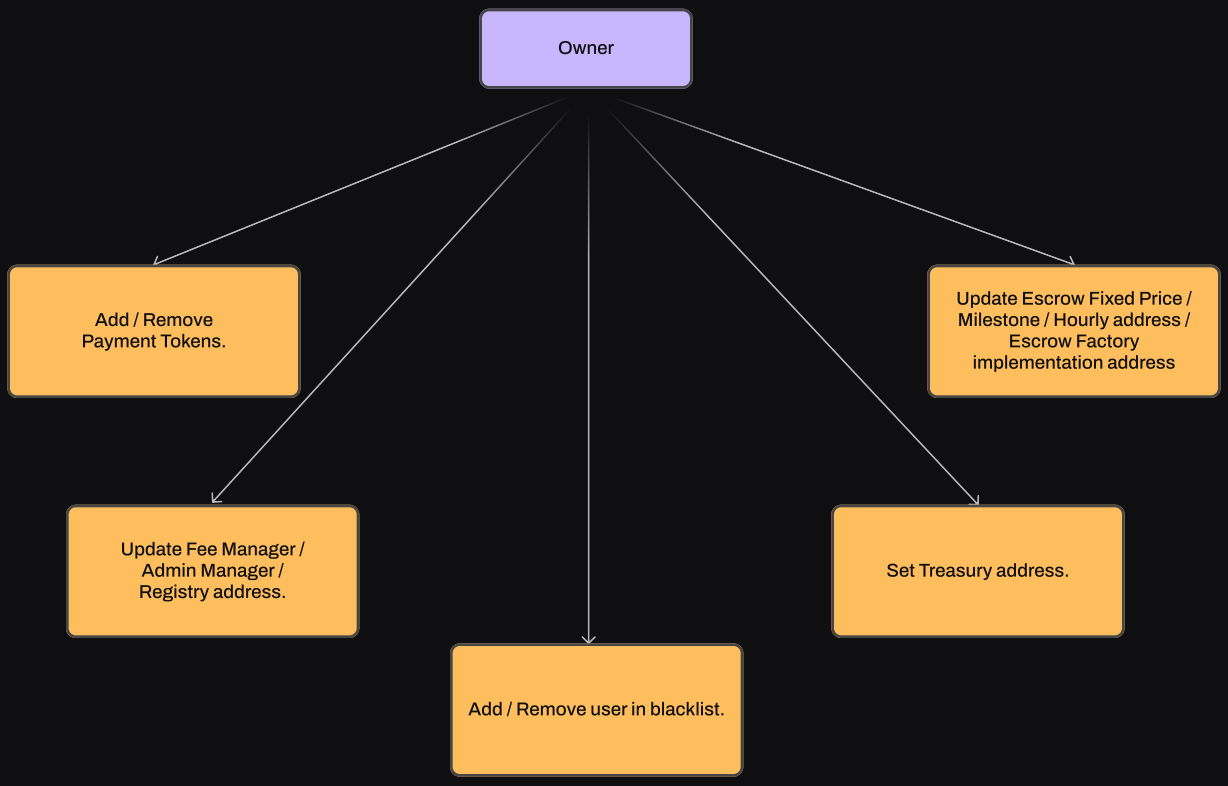


ESCROWREGISTRY

DEPLOYMENT

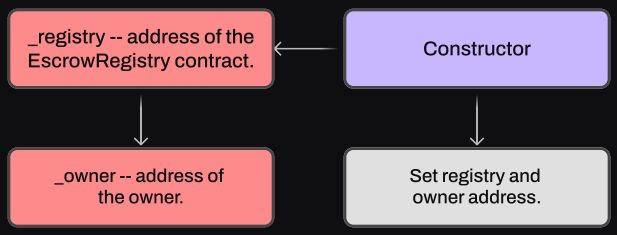


OWNER SETTERS

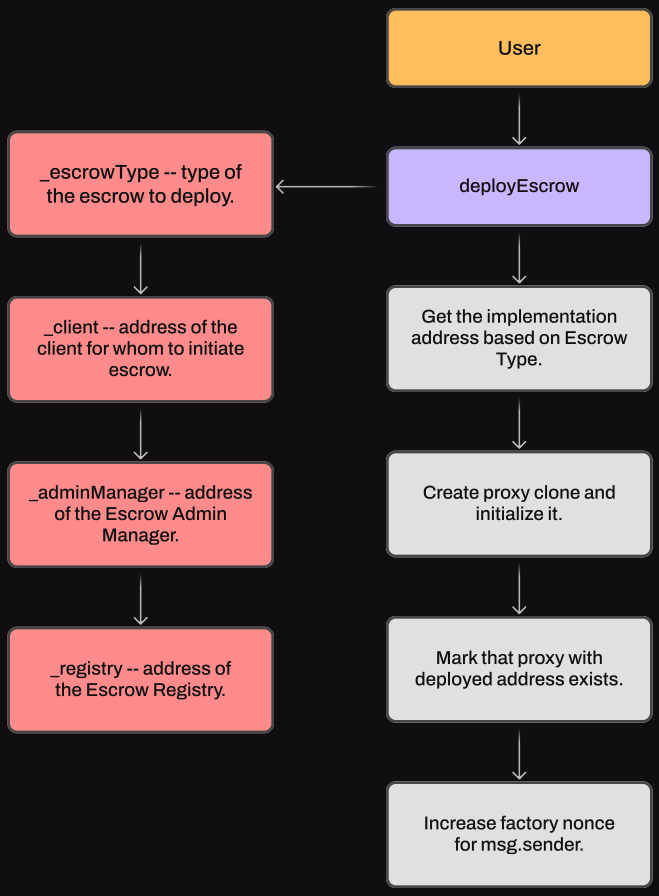


ESCROWFACTORY

DEPLOYMENT

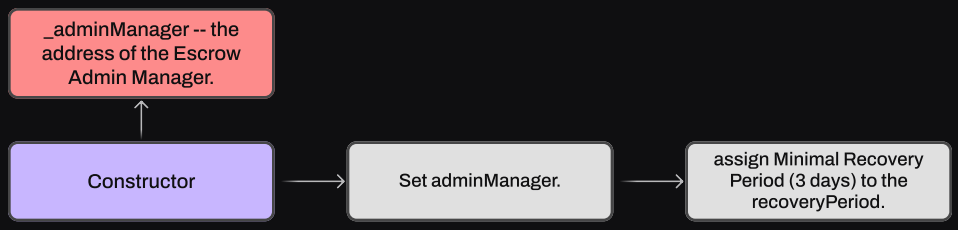


ESCROW CREATION



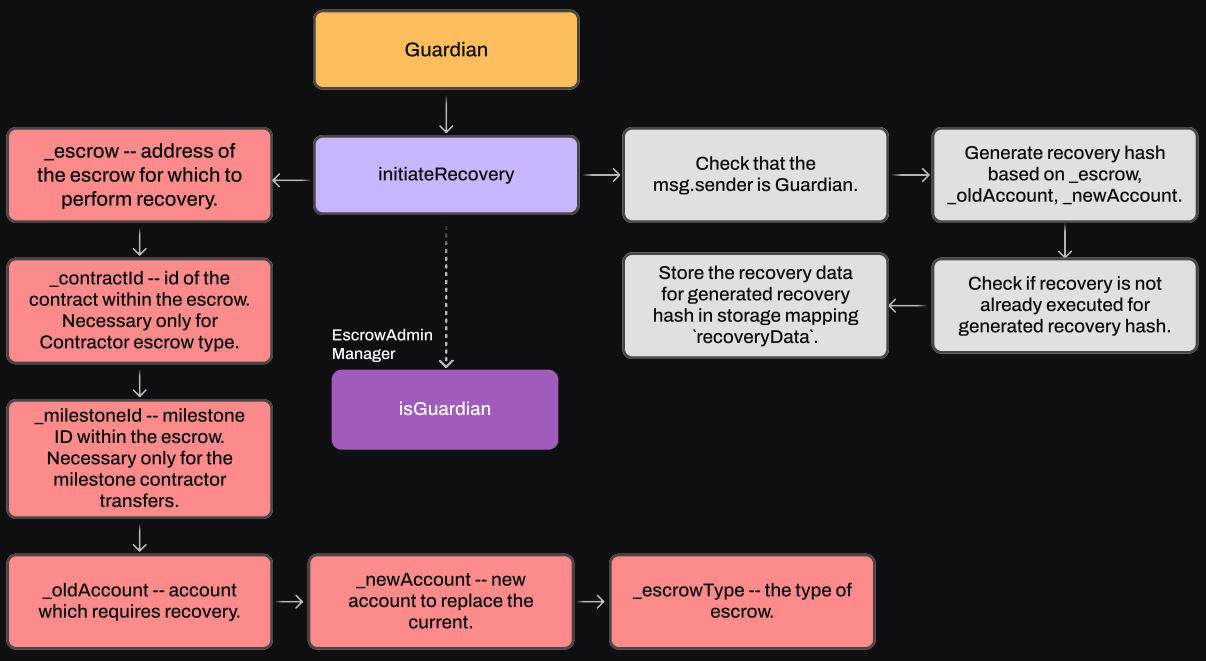
ESCROWACCOUNTRECOVERY

DEPLOYMENT



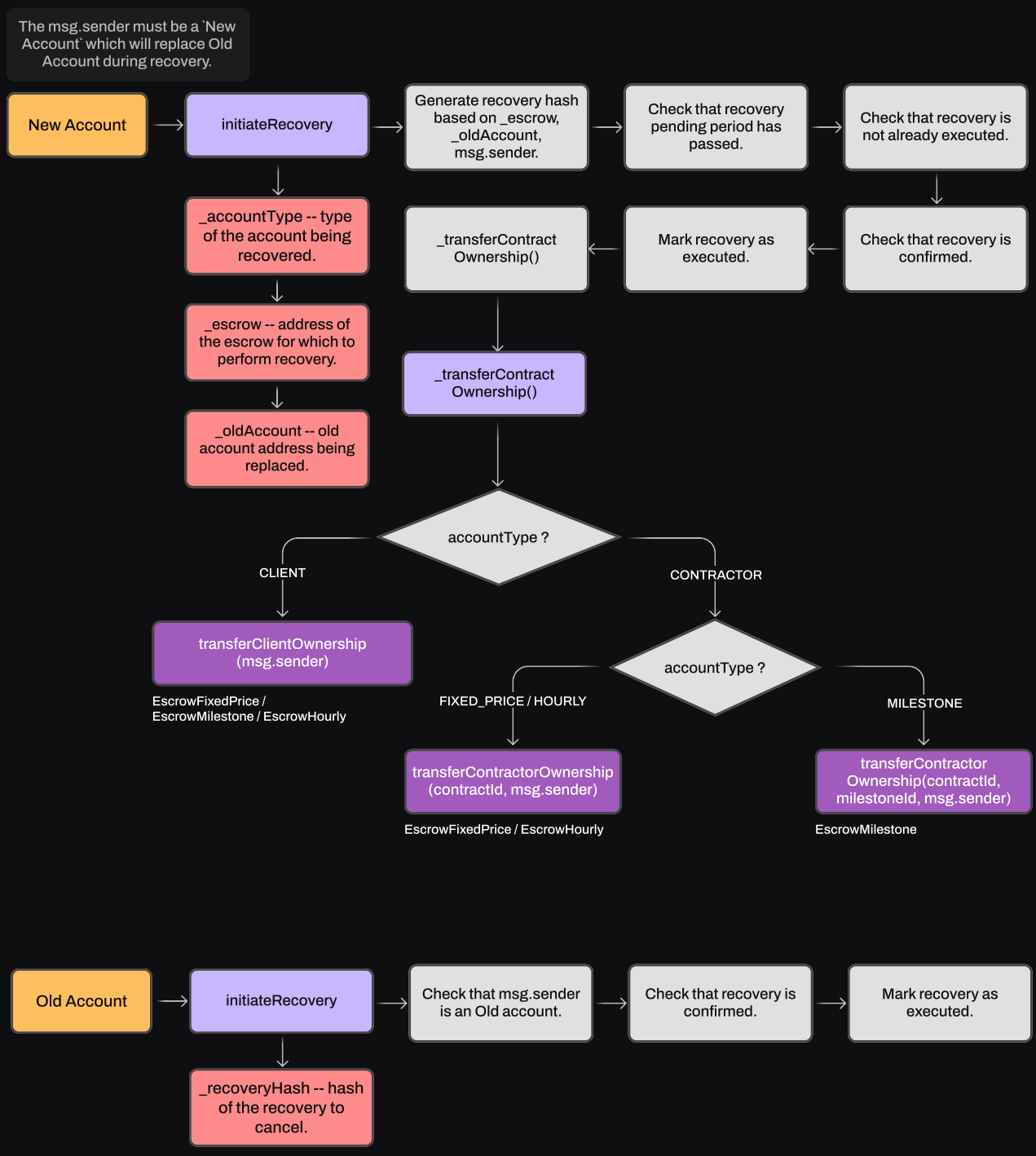
SETTERS

RECOVERY FLOW



ESCROWACCOUNTRECOVERY

RECOVERY FLOW



Complete Analysis

STANDARD CHECKLIST / VULNERABLE AREAS

Storage structure and data modification flow	Pass
Access control structure, roles existing in the system	Pass
Public interface and restrictions based on the roles system	Pass
General Denial Of Service (DOS)	Pass
Entropy Illusion (Lack of Randomness)	N/A
Order-dependency and time-dependency of operations	Pass
Accuracy loss, incorrect math/formulas other violated operations with numbers	Pass
Validation of function parameters, inputs validation	Pass
Asset management, funds flow and assets conversions	Pass
Signatures replay and multisig schemes security	Pass
Asset Security (backdoors connected to underlying assets)	Pass
Incorrect minting, initial supply or other conditions for assets issuance	Pass
Global settings mis-using, incorrect default values	Pass
Violated communication between components/modules, broken co-dependencies	Pass
3rd party dependencies, used libraries and packages structure	Pass
Single point of failure	Pass
Centralization risk	Pass
General code structure checks and correspondence to best practices	Pass
Language-specific checks	Pass

CRITICAL-1 Resolved

ARBITRARY ACCOUNT TYPE PASSED WHICH CAN ALLOW TO STEAL CLIENT/CONTRACTOR ROLE AND FUNDS

EscrowAccountRecovery.sol: executeRecovery().

The function executeRecovery() can be called by a new account of client or contractor after the Guardian has initiated the recovery process. One of the function parameters, _accountType, defines if the client or contractor role of the escrow or contract is recovered. However, this parameter is not validated (for example, not included in the recovery hash set by the Guardian). As a result, the new account might steal the opposing role. For example, the recovery might have been initiated to recover the client role, however, the new account specified a “Contractor” recovery type or vice versa. By doing so, a new account can obtain a role for which he is not designated and steal either withdrawable funds of the client or claimable funds of the contractor. Though the new account is specified by the guardian, it can still perform malicious actions since the new account is supposed to be a new address of an existing client or contractor.

RECOMMENDATION:

Set the account type of the recovery in the function initiateRecovery(). This way only authorized Guardian accounts can specify it, excluding the ability for users to pass it arbitrarily.

POST-AUDIT:

accountType is now encoded in the recovery hash, thus an arbitrary value can't be passed.

CRITICAL-2 Resolved

ARBITRARY PARAMETERS CAN BE PROVIDED

1. EscrowFactory.sol: deployEscrow().

Function allows anyone to deploy an Escrow and pass arbitrary addresses of module smart contracts such as admin manager and registry. These smart contracts are used in the escrow for several purposes such as:

- Validating the access to restricted functions.
- Provide address of Fee Manager and Treasury necessary to calculate and transfer valid fees.

As a result, users can provide invalid parameters to gain advantage within their Escrow. However, such an escrow will be registered in the Factory and be considered a valid Escrow displayable for contractors until it is known that passed parameters are invalid.

2. EscrowFixedPrice.sol, EscrowHourly.sol, EscrowMilestone.sol: deposit(), "_deposit.feeConfig".

Function allows clients to specify arbitrary fee config, including an option "NO_FEES" option. This way, users can always specify this option avoiding the protocol fees.

RECOMMENDATION:

Either allows only authorized users to set the specified parameters or validate them in some other way (for example, by signing these parameters by authorized admin and checking the validity of signature).

POST-AUDIT:

1) Addresses of AdminManager and Registry are now stored in the contract's storage.
2) Parameters of each deposit are now signed by the Admin and validated during the deposit. Thus, arbitrary fee config can no longer be passed.

MEDIUM-1 Resolved

CONTRACTOR'S DATA IS NOT VALIDATED TO CORRESPOND TO THE CONTRACTOR

EscrowFixedPrice.sol, EscrowMilestone.sol: submit().

During the function "deposit()", the client is able to specify a contractor's data hash. After that, in the function "submit()", the contractor can provide data and salt to generate and validate the contractor's data hash. Though only the contractor is supposed to contain the data (since the smart contract only stores hash), the data can still be visible in the function parameters. A malicious actor can potentially monitor the memory pool, see the contractor's transaction and his data and frontrun this transaction with his own transaction with the same data.

RECOMMENDATION:

Validate that provided data corresponds to the contractor (by storing which data corresponds to which contractors in advance or by using signatures signed by authorized users of the protocol).

POST-AUDIT:

Input parameters are now validated to be signed by the Admin. The signature includes both msg.sender and contractorData, thus, malicious actors can't use someone else's data.

MEDIUM-2 Resolved

NEW STATUS MIGHT NOT BE EQUAL TO STATUS BEFORE REFUND WAS REQUESTED

EscrowFixedPrice.sol, EscrowHourly.sol, EscrowMilestone.sol: cancelReturn().

When the client calls the function cancelReturn(), he specifies a new status of the contract. However, the status is arbitrary and might not correspond to the previous status of the contract. This can potentially break the logic of the contract, since it relies on a strict order of status changes.

RECOMMENDATION:

Add a validation so that the client can't pass an arbitrary status.

POST-AUDIT:

Previous status is now stored during requesting of return and set back in cancelReturn()

LOW-1 Resolved

USERS CAN BE BLACKLISTED BEFORE THEY CLAIMED/WITHDREW FUNDS

EscrowFixedPrice.sol, EscrowHourly.sol, EscrowMilestone.sol: claim(), withdraw().

Function claim() allows contractors to claim their payment while withdraw() allows clients to collect their funds back. Both functions check if the msg.sender is in blacklist. Thus, the users might be blocked before they were able to collect their funds back. The issue is marked as low since only the admin can add users to the blacklist.

RECOMMENDATION:

Ensure that users can claim their funds without being blocked due to blacklist.

POST-AUDIT:

User can now claim/withdraw even if blacklisted.

LOW-2 Verified

ADMIN MANAGER AND REGISTRY MUST BE UPDATED IN ALL THE ESCROWS SEPARATELY.

EscrowFixedPrice.sol, EscrowHourly.sol, EscrowMilestone.sol: updateRegistry(), updateAdminManager().

Functions allow updating the registry and admin manager. However, in case these addresses have to be updated in all the escrows, this might consume a great amount of gas. Though a multicall smart contract can be used, it won't decrease the gas spendings to updated addresses in all the escrows.

RECOMMENDATION:

Store the addresses of admin manager and registry in a single point of the protocol (for example, in the Factory).

POST-AUDIT:

The client has verified that current approach is more preferable as updating these addresses is rare event, it minimizes users' gas costs as addresses are stored in the storage of each escrow and such approach provides a clear separation of responsibilities.

INFO-1 Resolved

OLD ACCOUNT CAN BLOCK THE RECOVERY

EscrowAccountRecovery.sol: cancelRecovery().

The function allows old account to cancel the recovery process. Though the logic of the recovery implies that the user should have lost access to an old account and thus, having the ability to cancel recovery proves that he still has access, such logic does not cover cases where the account of the user was exploited. In this case, the recovery might be initiated to remove access to the Escrow for malicious actors who have access to the user's account. However, the malicious actor might use the function cancelRecovery(), thus, canceling any recovery process.

Issue is marked as info since it is unknown if this logic should cover key leaking cases and the criticality may be increased based on the feedback.

RECOMMENDATION:

Ensure that malicious actors can't block the recovery in case of private key leaking of an old account (for example by using a blacklist).

POST-AUDIT:

msg.sender is now validated not to be in blacklist, thus, blacklisted old account can't block the recovery.

INFO-2 Resolved

ANYONE CAN CREATE ESCROW FOR ANY CLIENT

EscrowFactory.sol: deployEscrow().

The comments suggest that the parameter "_client" is an address which initiates the contract and should be connected with the msg.sender. However, this parameter is not validated to be equal to msg.sender or be connected with it. This allows anyone to create multiple escrows for a client without his permission. As a result, the client might see multiple escrows on the front-end created for him. And that opens a phishing possibility thus increasing the negative risk for the client.

The issue is marked as Info, as it currently cannot be classified without additional information from the protocol team.

RECOMMENDATION:

Ensure that malicious actors can't block the recovery in case of private key leaking of an old account (for example by using a blacklist).

POST-AUDIT:

msg.sender is now validated not to be in blacklist, thus, blacklisted old account can't block the recovery.

INFO-3 Resolved

BEST PRACTICES AND CODE STYLE VIOLATIONS, OPTIMIZATIONS

1. No function to withdraw ETH.

EscrowAdminManager.sol, EscrowFeeManager, EscrowRegistry, EscrowFactory.

Contracts inherit OwnedRoles/OwnedThreeStep smart contract by Solbase which have a payable function. Thus, the ETH can accidentally occur on the balance of smart contracts.

2. Redundant struct field.

EscrowAccountRecovery.sol: struct “RecoveryData”.

The field “confirmed” of struct “RecoveryData” is redundant. The value of this field in each created struct is assigned to true in the function “initiateRecovery()” and is never changed. It is only used in the function “executeRecovery()”. However, this check can be replaced with a similar check (for example, like in function “cancelRecovery()” or by validating one of the address fields not to be zero address).

3. Missing validations.

MoonshotToken.sendingToPairNotAllowed

a. EscrowAccountRecovery.sol: initiateRecovery().

- _escrow – missing zero-address validation.
- _contractId, _milestoneId - check validity of passed IDs (their existence and necessity since milestone is only needed for a EscrowMilestone). It should also be noted that right now it is possible to transfer ownership of uninitialized contract ID.

b. EscrowFeeManager.sol: setUserSpecificFees(), setInstanceFees(), setContractSpecificFees(), _setDefaultFees().

- _coverage, _claim - both fee percentage is checked not to exceed MAX_BPS (100%). However, having the ability to set fees up to 100% percent is not a good practice as fees can be set to great value due to human error or in case of private key leaking. In such cases it is recommended to introduce a certain boundary for fee values.

c. EscrowAccountRecovery.sol: updateRecoveryPeriod().

- _recoveryPeriod -- should be validated for maximum boundary in order to avoid setting large values which can potentially block the recovery mechanism.

4. EscrowAccountRecovery.sol: _transferContractOwnership().

There is an “if else” statement that checks “escrowType”. This statement consists of 3 branches for each escrow type. However, it can be simplified since the branches only differ by the interface used to call the function of the escrow. Since all the escrows have the same function with the same signature, the complex “if else” statement can be removed using a common interface for all escrows instead.

5. Initializable contract can be re-used.

EscrowFixedPrice.sol, EscrowHourly.sol, EscrowMilestone.sol.
All smart contracts use initialize() functions instead of constructor. The validation that the contract is initialized only once is performed directly in the contract. Instead it is recommended to inherit Initializable smart contract by OpenZeppelin which performs all the necessary validations.

Post-audit:

All the points were fixed.

Disclaimer

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.