Zellic

Prepared for
Gabriel Shapiro
Michael Erisman
MetaLex

Prepared by
Weipeng Lai
Varun Verma
Zellic

October 16, 2025

cyberRaise
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for MetaLex from October 7th to October 13th, 2025. During this engagement, Zellic reviewed cyberRaise's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are user funds secure throughout the process?
- Are there any chances of deadlocks or exploits?
- Is the storage pattern free from issues when upgrading?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

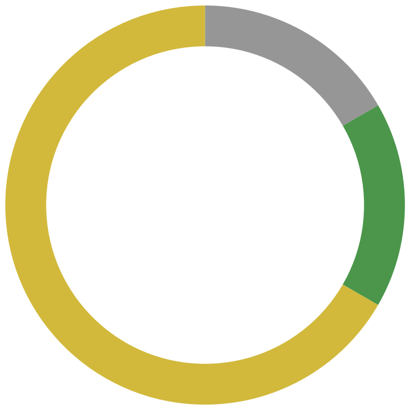
1.4. Results

During our assessment on the scoped cyberRaise contracts, we discovered six findings. No critical issues were found. Four findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of MetaLex in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	4
🟩 Low	1
⬜ Informational	1



2. Introduction

2.1. About cyberRaise

MetaLex contributed the following description of cyberRaise:

> MetaLeX — automated onchain legal startup seed rounds.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

cyberRaise Contracts

Type	Solidity
Platform	EVM-compatible

Target	cybercorps-contracts
Repository	https://github.com/MetaLex-Tech/cybercorps-contracts ↗
Version	`2a7d5dc5cd1889302928ea09dfbb35b9326bdd2d`

Programs	`RoundManager.sol`
	`DealManager.sol`
	`libs/LexScroWLite.sol`
	`RoundManagerFactory.sol`
	`DealManagerFactory.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.4 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Weipeng Lai
Engineer
weipeng.lai@zellic.io

Varun Verma
Engineer
varun@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 7, 2025	Kick-off call
October 7, 2025	Start of primary review period
October 13, 2025	End of primary review period

3. Detailed Findings

3.1. Unvalidated agreement metadata in FCFS EOIs

Target	RoundManager		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	Medium

Description

The `submitEOI` function in RoundManager forwards the caller-supplied `globalValues` and `partyValues` directly into the agreement without any validation or sanitization. In first-come, first-served (FCFS) rounds, the contract immediately finalizes by calling `allocate`, which triggers `signContractWithEscrow`, so the CyberCorp never has a chance to review the submitted metadata.

```
function submitEOI(
    bytes32 roundId,
    EOI memory eoi,
    string[] memory globalValues,
    string[] memory partyValues,
    bytes memory signature,
    uint256 salt,
    address[] memory conditions,
    bytes32 secretHash
) external returns (bytes32 agreementId, uint256 tokenId) {
    // [...]
    (agreementId, tokenId) = RoundManagerStorage.submitEOI(
        LexScrowStorage.lexScrowStorage(),
        roundId,
        eoi,
        globalValues,
        partyValues,
        signature,
        salt,
        conditions,
        secretHash
    );
    // [...]
    if (round.roundType == RoundType.FCFS) {
        this.allocate(agreementId, eoi.maxAmount);
    }
}
```

An investor can therefore record arbitrary statements (for example, claiming a 10,000 USD commitment while only escrowing 100 USD) that the CyberCorp is forced to cosign, producing a finalized agreement whose on-chain terms do not match the funds actually committed.

Impact

Because FCFS rounds autofinalize, any participant can craft misleading or abusive metadata without CyberCorp review.

Recommendations

We recommend validating agreement metadata before acceptance by allowing the CyberCorp to configure an allowlist of permitted `globalValues` entries and making `submitEOI` reject any submission that includes values outside that list.

Remediation

This issue has been acknowledged by MetaLex.

MetaLex provided the following response to this finding:

> The final legal agreements will be structured in such a way that will protect from malicious input attempts. The deal will follow the investment amount, subject to the round min/max, and the tokenized certificates will only reflect the actual paid amount. Founders will have the ability, and legal backing, to handle someone intentionally putting false text fields into the legal agreement.

3.2. Escrow counterparty misassignment

Target	DealManager		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `signDealAndPay` and `signAndFinalizeDeal` functions in DealManager pass `msg.sender` to `updateEscrow`, causing the escrow to store the transaction caller as the counterparty instead of the authenticated signer.

```
function signDealAndPay(
    address signer,
    bytes32 agreementId,
    bytes memory signature,
    string[] memory partyValues,
    bool _fillUnallocated,
    string memory name,
    string memory secret
) public {
    // [...]
    updateEscrow(agreementId, msg.sender, name);
    // [...]
}

function signAndFinalizeDeal(
    address signer,
    bytes32 agreementId,
    string[] memory partyValues,
    bytes memory signature,
    bool _fillUnallocated,
    string memory name,
    string memory secret
) public {
    // [...]
    updateEscrow(agreementId, msg.sender, name);
    // [...]
}

function updateEscrow(bytes32 agreementId, address counterParty,
    string memory buyerName) internal {
```

```
    Escrow storage escrow = LexScrowStorage.getEscrow(agreementId);
    escrow.counterParty = counterParty;
    [...]
}
```

When the deal finalizes, the corp assets transfer to the caller instead of the authenticated signer.

```
function finalizeEscrow(bytes32 agreementId) internal {
    // [...]
    for(uint256 i = 0; i < escrow.corpAssets.length; i++) {
        if(escrow.corpAssets[i].tokenType == TokenType.ERC20) {

    IERC20(escrow.corpAssets[i].tokenAddress).safeTransfer(escrow.counterParty,
    escrow.corpAssets[i].amount);
        }
        else if(escrow.corpAssets[i].tokenType == TokenType.ERC721) {

    IERC721(escrow.corpAssets[i].tokenAddress).safeTransferFrom(address(this),
    escrow.counterParty, escrow.corpAssets[i].tokenId);
        }
        else if(escrow.corpAssets[i].tokenType == TokenType.ERC1155) {

    IERC1155(escrow.corpAssets[i].tokenAddress).safeTransferFrom(address(this),
    escrow.counterParty, escrow.corpAssets[i].tokenId,
    escrow.corpAssets[i].amount, "");
        }
    }
}
```

Impact

An attacker who observes a legitimate investor's `signDealAndPay` or `signAndFinalizeDeal` transaction in the mempool can front-run the transaction and submit the same call with the victim's signature. Although the attacker must provide payment, they can receive the corp assets without signing the agreement.

Recommendations

We recommend setting the escrow counterparty to the validated signer.

```
updateEscrow(agreementId, msg.sender, name);
updateEscrow(agreementId, signer, name);
```

Remediation

This issue has been acknowledged by MetaLex, and a fix was implemented in commit `4c808dc8` ↗.

3.3. Malicious registry signing blocks deal participation

Target	DealManager		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The DealManager contract expects deals to be signed through `signDealAndPay` or `signAndFinalizeDeal`, which calls `signContractFor` in CyberAgreementRegistry to sign the agreement:

```
function signDealAndPay(
    address signer, bytes32 agreementId,
    bytes memory signature, string[] memory partyValues,
    bool _fillUnallocated, string memory name,
    string memory secret
) public {
    // [...]
    ICyberAgreementRegistry(LexScrowStorage.getDealRegistry())
        .signContractFor(signer, agreementId, partyValues,
            signature, _fillUnallocated, secret);
    // [...]
}

function signAndFinalizeDeal(
    address signer, bytes32 agreementId, string[] memory partyValues,
    bytes memory signature, bool _fillUnallocated,
    string memory name, string memory secret
) public {
    // [...]
    if(!ICyberAgreementRegistry(LexScrowStorage.getDealRegistry())
        .hasSigned(agreementId, signer))
        ICyberAgreementRegistry(LexScrowStorage.getDealRegistry())
            .signContractFor(signer, agreementId, partyValues,
                signature, _fillUnallocated, secret);
    // [...]
}
```

However, any party can call `signContractFor` in CyberAgreementRegistry directly. For an open deal, one signature slot in the registry remains available for a signer. When the first signer claims the slot, other investors cannot sign the contract and participate in the deal.

```
function signContractFor(
    address signer,
    bytes32 contractId,
    string[] memory partyValues,
    bytes calldata signature,
    bool fillUnallocated, // to fill a 0 address or not
    string memory secret
) public {
    // [...]
    if (!isParty(contractId, signer)) {
        if (
            agreementData.secretHash > 0 &&
            keccak256(abi.encode(secret)) != agreementData.secretHash
        ) revert InvalidSecret();
        // Not a named party, so check if there's an open slot
        uint256 firstOpenPartyIndex = getFirstOpenPartyIndex(contractId);
        if (firstOpenPartyIndex == 0 || !fillUnallocated)
            revert NotAParty();
        // There is a spare slot, assign the sender to this slot.
        agreementData.parties[firstOpenPartyIndex] = signer;
        agreementsForParty[agreementData.parties[firstOpenPartyIndex]].push(
                contractId
            );
    }
    // [...]
    if (totalSignatures == agreementData.parties.length) {
        if (agreementData.finalizer == address(0)) {
            agreementData.finalized = true;
            emit ContractFinalized(contractId, msg.sender, timestamp);
        }

        emit ContractFullySigned(contractId, timestamp);
    }
}
```

A malicious counterparty can sign the deal directly through the registry, consuming the limited signature slots without performing the escrow payment. Because the slot is filled, legitimate investors cannot participate in the deal.

Impact

For open deals, an attacker can prevent investors from participating in deals by consuming the signature slot in the registry.

Recommendations

Restrict `signContractFor` so that only DealManager or designated contracts can invoke it.

Remediation

This issue has been acknowledged by MetaLex, and a partial fix was implemented in commit 24386429 ↗.

MetaLex provided the following response to this finding:

> We will check that sender matches the signature sent in so that someone can't steal a signature and front run, but we don't want "open deal" agreements to be restricted to finalizers as they are used for other use cases. If a deal gets griefed, we would suggest to send a closed offer to the party attempting to invest.

3.4. Dust loss in RoundManager allocation

Target	RoundManagerStorage		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `allocate` function in RoundManager computes `units` and `investmentUSD` through integer division. Remainders from `allocatedAmount` that do not divide evenly by `pricePerUnit` or `10 ** paymentDecimals` are not processed.

The contract records `unitsRepresented` and `investmentAmountUSD` based on these truncated values while maintaining the original `allocatedAmount` for treasury accounting. The remainder is neither refunded nor accounted for elsewhere.

```
function allocate(
    LexScrowStorage.LexScrowData storage ls,
    bytes32 agreementId,
    uint256 allocatedAmount
)
external // Use external library to save space
returns (uint256 tokenId, uint256[] memory certIds, uint256 refund) {
    // [...]
    uint256 units = allocatedAmount / round.pricePerUnit;
    uint8 paymentDecimals = IERC20Metadata(round.paymentToken).decimals();
    uint256 investmentUSD = allocatedAmount / (10 ** paymentDecimals);
    // [...]
    CertificateDetails memory details = CertificateDetails({
        signingOfficerName: officerName,
        signingOfficerTitle: officerTitle,
        investmentAmountUSD: investmentUSD,
        issuerUSDValuationAtTimeOfInvestment: round.valuation,
        unitsRepresented: units,
        legalDetails: round.legalDetails,
        extensionData: round.extensionData
    });
    // [...]
    refund = escrow.buyerAssets[0].amount - allocatedAmount;
    escrow.buyerAssets[0].amount = allocatedAmount;
    // [...]
}
```

Impact

Investors lose funds when payments are not divisible by `pricePerUnit` or token decimals. Certificates reflect fewer units and lower USD amounts than paid.

Recommendations

Calculate and refund remainders. After computing `units = allocatedAmount / pricePerUnit`, calculate `uint256 remainder = allocatedAmount - (units * pricePerUnit)`. Apply the same calculation for USD amounts. Return remainders to investors before finalizing the escrow.

Remediation

This issue has been acknowledged by MetaLex, and fixes were implemented in the following commits:

- `8c626f73` ↗
- `129396f3` ↗
- `4cb9723c` ↗

The dust amount resulting from rounding is now refunded.

Note: Although `usedAmount` (rather than `allocatedAmount`) now represents the actual raised amount, the `allocate` function in the RoundManager contract continues to enforce the minimum amount requirement against `allocatedAmount` instead of `usedAmount`. MetaLex provided the following rationale for this design decision:

MetaLex provided the following response to this finding:

> The maximum rounding down is 1 unit of precision of the payment token. (0.000001 for usdc) If the founder wants to allocate at the exact minAmount of the eoi, we would not want the contract to revert for that reason for usability/UX. We are okay acknowledging that this small amount may be under the min set – the minimum here is mainly to avoid very small nuisance investments.

3.5. DealManager could store different `counterPartyValues` from those in the agreement

Target	DealManager		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `signAndFinalizeDeal` function in DealManager skips calling `signContractFor` when the signer has already signed in the registry:

```
function signAndFinalizeDeal(
    address signer,
    bytes32 agreementId,
    string[] memory partyValues,
    bytes memory signature,
    bool _fillUnallocated,
    string memory name,
    string memory secret
) public {
    // [...]
    if(!ICyberAgreementRegistry(LexScrowStorage.getDealRegistry())
       .hasSigned(agreementId, signer)) {
        ICyberAgreementRegistry(LexScrowStorage.getDealRegistry())
            .signContractFor(signer, agreementId, partyValues,
                        signature, _fillUnallocated, secret);
    }
    updateEscrow(agreementId, msg.sender, name);
    if(!conditionCheck(agreementId)) revert AgreementConditionsNotMet();
    handleCounterPartyPayment(agreementId);
    finalizeDeal(agreementId);
}
```

This logic allows the counterparty to create inconsistent `counterPartyValues` between DealManager and CyberAgreementRegistry when a deal leaves `counterPartyValues` unset. The attack proceeds as follows:

1. The counterparty calls `signContractFor` in CyberAgreementRegistry directly with one set of `partyValues`.

2. The counterparty then calls `signAndFinalizeDeal` in DealManager with a different set of `partyValues`.

Because `signAndFinalizeDeal` skips calling `signContractFor` when the signer has already signed in the registry, the function proceeds to update its local storage with the new `partyValues` without revalidation. This creates a state divergence; the `partyValues` stored in DealManager differ from those stored in CyberAgreementRegistry, leaving the two contracts with inconsistent records of `counterPartyValues` for the same agreement.

Impact

The on-chain registry and DealManager can disagree on the `counterPartyValues`. Downstream systems relying on `counterPartyValues` in DealManager will read manipulated data, enabling misreporting or bypassing business logic that expects consistency with the registry.

Recommendations

When `signAndFinalizeDeal` detects an existing signer, fetch the signer's `partyValues` from CyberAgreementRegistry and enforce equality before finalizing.

Remediation

This issue has been acknowledged by MetaLex, and fixes were implemented in the following commits:

- `8e8b6bba` ↗
- `854c16b2` ↗

3.6. Redundant assignment

Target	DealManager		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `initialize` function in DealManager sets `corp` and `dealRegistry` twice — directly through `LexScrowStorage.set*` and again within `__LexScroWLite_init`.

```
function initialize(address _auth, address _corp, address _dealRegistry,
    address _issuanceManager, address _upgradeFactory) public initializer {
    // [...]
    LexScrowStorage.setCorp(_corp);
    LexScrowStorage.setDealRegistry(_dealRegistry);
    // [...]

    // [...]
    __LexScroWLite_init(_corp, _dealRegistry);
    // [...]
}

function __LexScroWLite_init(address _corp, address _dealRegistry)
    internal onlyInitializing {
    LexScrowStorage.setCorp(_corp);
    LexScrowStorage.setDealRegistry(_dealRegistry);
}
```

These redundant assignments waste gas by writing the same values to storage twice.

Impact

The function consumes additional gas for redundant storage writes.

Recommendations

Remove the direct `setCorp` and `setDealRegistry` calls from `initialize`. Rely solely on `__LexScroWLite_init` to set these values.

Remediation

This issue has been acknowledged by MetaLex, and a fix was implemented in commit `291e723d` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Insufficient input validation in deal and round creation

The DealManager and RoundManager contracts lack adequate input validation when creating deals or rounds. For example, `proposeDeal` in DealManager does not enforce that the `_parties` array contains exactly two elements, and `proposeAndSignDeal` in DealManager does not verify that the `proposer` parameter matches `_parties[0]`.

Although these functions use the `onlyOwner` modifier, we recommend adding sanity checks. These checks help catch incorrect parameters early and prevent invalid agreements when CyberCorp accidentally supplies incorrect values during deal or round creation.

4.2. Missing explicit agreement existence check in `voidExpiredDeal`

The `voidExpiredDeal` function in DealManager assumes that a deal exists for the supplied `agreementId` but does not explicitly verify this. The existence check occurs only implicitly within the `voidContractFor` function.

```
function voidExpiredDeal(bytes32 agreementId, address signer,
    bytes memory signature) public nonReentrant {
    // Check: status
    Escrow storage deal = LexScrowStorage.getEscrow(agreementId);
    if (block.timestamp <= deal.expiry) revert DealNotExpired();

    // Effect: update status

    ICyberAgreementRegistry(LexScrowStorage.getDealRegistry()).voidContractFor(
        agreementId, signer, signature);
    for(uint256 i = 0; i < deal.corpAssets.length; i++) {
        if(deal.corpAssets[i].tokenType == TokenType.ERC721) {
            DealManagerStorage.getIssuanceManager().voidCertificate(
                deal.corpAssets[i].tokenAddress,
                deal.corpAssets[i].tokenId
            );
        }
    }

    if(deal.status == EscrowStatus.PAID)
```

```
            // Interaction: payment
            voidAndRefund(agreementId);
        else if(deal.status == EscrowStatus.PENDING)
            // Effect: update status
            voidEscrow(agreementId);
    }
```

If a future upgrade modifies `voidContractFor` to skip processing rather than reverts for a nonexistent `agreementId`, an attacker could call `voidExpiredDeal` with an `agreementId` that has not yet been created and mark the deal as void in the contract.

Therefore, we recommend adding an explicit existence check for the `agreementId` parameter in `voidExpiredDeal`.

4.3. The proxy is deployed uninitialized

The `DealManagerFactory.deployDealManager` deploys an `ERC1967Proxy` without initialization calldata, so the proxy is left uninitialized.

```
function deployDealManager(bytes32 _salt) public returns (address) {
    if (_salt == bytes32(0)) revert InvalidSalt();

    // Create proxy deployment bytecode
    bytes memory proxyBytecode = _getBytecode();

    // Deploy using CREATE2
    address dealManagerProxy = Create2.deploy(0, _salt, proxyBytecode);

    if(dealManagerProxy == address(0)) revert DeploymentFailed();

    emit DealManagerDeployed(dealManagerProxy, DealMan-
    ager(DealManagerFactoryStorage.getRefImplementation()).DEPLOY_VERSION());
    return dealManagerProxy;
}

function _getBytecode() private view returns (bytes memory bytecode) {
    bytes memory sourceCodeBytes = type(ERC1967Proxy).creationCode;
    bytecode = abi.encodePacked(sourceCodeBytes,
    abi.encode(DealManagerFactoryStorage.getRefImplementation(), ""));
}
```

This is safe only if the deployment flow wraps `deployDealManager` and the proxy's `initialize` call within a single atomic transaction.

If `deployDealManager` can be invoked independently, an attacker could front-run the initialization by calling `initialize` on the newly deployed proxy first, thereby configuring it with malicious parameters.

The same concern also exists in RoundManagerFactory.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: CyberCorp round and deal manager system

Description

The CyberCorp system is built around two main flows that share the same escrow backbone: fundraising rounds and one-off deals. Both rely on LexScroWLite, the core escrow engine that handles payment intake, state transitions, fee distribution, and asset delivery.

This is a brief overview of aspects of the **escrow engine** (LexScroWLite).

- It controls the full life cycle of an agreement through clear states: `PENDING` → `PAID` → `FINALIZED` or `VOIDED`.
- The `handleCounterPartyPayment` pulls buyer assets (ERC-20/721/1155) into escrow and marks it as paid.
- The `finalizeEscrow` function first updates the status to `FINALIZED` then sends ERC-20 tokens to `companyPayable` and fees to `getPlatformPayable`. Corp assets (certificates) go to the escrow's counterparty.
- The `voidAndRefund` safely reverses the flow by setting the status to `VOIDED` before refunding buyer assets.
- Conditions are enforced through `conditionCheck`, looping through each attached ICondition contract and requiring all to pass.
- Reentrancy protection and strict ordering (state update before external calls) make double-spends and recursive refunds impossible.

This is a brief overview of aspects of **RoundManager**.

- It manages multi-investor fundraising rounds.
- Rounds are created via `createRound`, which requires a valid EIP-712 signature from an authorized officer (`_verifyEscrowedSignature`) binding all parameters — pricing, caps, time window, token, and corp address.
- Investors submit EOIs through `submitEOI`, which validates timing and ticket size, records escrow data, and pulls payment immediately. For FCFS rounds, allocation is triggered automatically.
- The `allocate` function enforces round validity, EOI freshness, cap limits, and correct status (`PAID`, not already allocated). It calculates the eligible allocation, refunds any excess payment, and finalizes escrow to transfer funds and certificates.
- Both `reject` and `recallEOI` handle refunds, requiring the escrow to be paid and unallocated before returning funds and voiding the record.

- Fees are computed with `computeFee(size)` using a factory-defined ratio capped at 100% (`BASIS_POINTS` = 10,000).
- Upgrades are restricted so that only the factory's reference implementation can be approved.

This is a brief overview of aspects of **DealManager**.

- It handles single deals with a similar escrow flow.
- The `proposeDeal` creates a contract in the registry, mints certificates via the issuance manager, and sets up the escrow with one ERC-20 asset flagged for fees.
- Both `signDealAndPay` and `signAndFinalizeDeal` process signatures and payments then update the escrow's counterparty, pull funds, and optionally finalize the agreement.
- There is a known issue. These functions use `updateEscrow(..., msg.sender, ...)`, meaning whoever submits the transaction becomes the counterparty. This opens a front-running vector where a relayer can finalize a deal with a valid signer's signature and receive the certificates. The attacker pays but gains ownership — so it is not a drain but still a logic flaw. The fix is to bind the counterparty to the `signer` or include the beneficiary in the signer's authorization as stated in finding 3.2.
- Deals can be finalized, voided, or refunded under strict conditions (e.g., `status == PAID`, not expired, all signatures complete).
- Governance, fee logic, and upgrade restrictions mirror those in RoundManager.

The **factories** (RoundManagerFactory and DealManagerFactory) do the following:

- Deploy new managers deterministically using `CREATE2` with the ERC-1967 proxy pattern
- Enforce that upgrades can only target the current reference implementation
- Restrict admin actions (fee ratio, payable address, reference impl) to the owner, with fee ratios capped at 100%

Invariants

- Escrow states are one-way and irreversible: `PENDING` → `PAID` → `FINALIZED/VOIDED`.
- Refunds can only occur once since refund functions require `status == PAID` and immediately mark the escrow as `VOIDED`.
- Finalization only occurs when the deal is not expired, all parties have signed, and all attached conditions pass.
- Payment flow is deterministic.
    - Company funds always go to `companyPayable`.
    - Platform fees always go to `getPlatformPayable`.
    - No other addresses can receive escrowed funds.
- Fees are bounded and safe.
    - Factories prevent ratios above 100%.
    - Checked arithmetic prevents overflow or underflow.

    - Only buyer tokens marked `isFee == true` have fees deducted.
- Round allocations are capped and consistent.
    - Respect investor deposit and min/max ticket, and raise cap.
    - Overdeposits are refunded before finalization.
    - The storage layer must update `escrow.buyerAssets[0].amount` after refunds to align with what is left to finalize (otherwise the transaction reverts but does not misdirect funds).
- Reject and recall flows.
    - Only valid if escrow is paid and unallocated.
    - Always void before refunding.
- All rounds require a verified EIP-712 signature from an authorized officer.
- Upgrades are gated to reference implementations via factory control.
- Refund tokens always match the payment token in the escrow (enforced in storage).

Test coverage

- RoundManager tests cover round creation, EIP-712 signature validation, EOI submission, allocation logic, cap enforcement, refunds, fee behavior, LexChex KYC checks, and upgrade permissions.
- DealManager tests go through end-to-end flows: proposal, signing, payment, finalization, voiding, and refunding — plus reverts for invalid signatures or deal states.
- Factories are tested for proper `CREATE2` deployment, deterministic address calculation, admin access control, and enforcement of fee-ratio limits and upgrade restrictions.
- CyberCorp integration tests confirm certificate issuance, LexChex condition enforcement, registry signing, secret handling, transfer hooks, and other cross-contract interactions.
- The current suite provides broad coverage of the intended flows, fund movements, and access control. What is missing is a negative case for the signer/beneficiary confusion: testing a scenario where `msg.sender` differs from the `signer` but still succeeds. Adding this test would highlight the vulnerability in finding 3.2 and confirm the fix once implemented.

Attack surface

The following outlines potential attack vectors through which adversaries could *attempt* to exploit the system. These represent surface-level entry points for adversarial behavior rather than identified vulnerabilities.

Relayer signature theft

Attackers can steal valid signatures from the mempool and front-run legitimate transactions in the deal-signing flows. By submitting the transaction first with a stolen signature, the attacker becomes the `escrow.counterParty` instead of the intended buyer. Certificates and assets get delivered to the attacker while the legitimate signer receives nothing. The attacker must provide payment, but this breaks the KYC chain, steals negotiated deal terms, and redirects ownership.

Factory initialization hijacking

New manager instances are deployed as uninitialized proxies through the factory system. Anyone monitoring the mempool can front-run the deployment and claim ownership by calling `initialize()` first with their own malicious auth contract. This gives complete control over the manager instance, access to all deals and escrows, and the ability to upgrade to backdoored implementations.

EIP-712 signature replay

Authorized officer signatures used in round creation and agreement signing could be captured and replayed. Without proper nonces and chain-specific bindings, these signatures can be reused across different chains, for different rounds, or after parameter changes. This enables unauthorized round creation, bypassing restrictions and reusing old approvals in unintended contexts.

Centralized control points

A compromised factory owner can push malicious reference implementations that all managers could upgrade to, manipulate fee ratios up to 100%, or redirect fee-payment addresses. Individual manager owners can be socially engineered into upgrading to backdoored code. The finalizer role has privileged powers to void agreements without requiring signatures, creating a centralized trust assumption.

6. Assessment Results

During our assessment on the scoped cyberRaise contracts, we discovered six findings. No critical issues were found. Four findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.