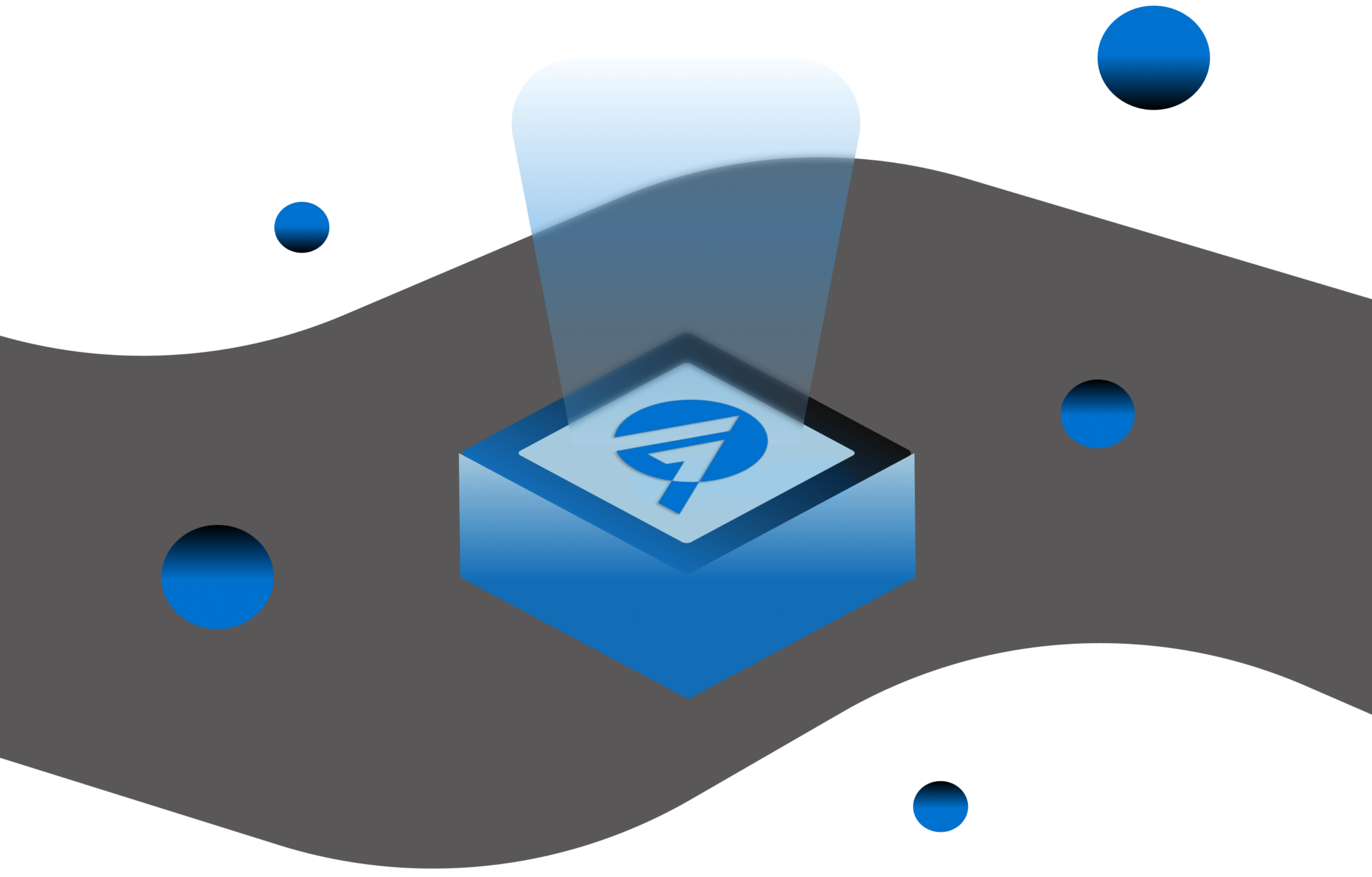
AUDITONE

Audit Report

14.12.2022

unicrow



Audit Report

by AuditOne

Smart Contract Security Analysis Report

Note: This report may contain sensitive information on potential vulnerabilities and exploitation methods. This must be referred internally and should be only made available to the public after issues are resolved (to be confirmed prior by the client and AuditOne).

Table of Contents

Introduction

Csanuragjain, Minhquanym, Tomo, Ak1 (nicknames), who are auditors at AuditOne, successfully audited the smart contracts of Unicrow. The audit has been performed using manual analysis. This report presents all the findings regarding the audit performed on the customer's smart contracts. The report outlines how potential security risks are evaluated. Recommendations on quality assurance and security standards are provided in the report.

Project Description

Unicrow is a censorship-resistant payment and escrow protocol that enables two parties to trade goods, assets, or services in the real world in a trust-minimized, efficient, and secure way.

Marketplaces and other platforms can integrate Unicrow for trades between their users without exposing or holding custody of their user's funds. The protocol reduces their operational expenditures and regulatory exposure thanks to its efficient, yet secure withdrawal mechanism, and non-custodial escrow design.

Unicrow also supports 3rd party arbitration while minimizing trust and security requirements on the arbitrators, and for situations where arbitration is not available, it introduces a unique and innovative challenge mechanism that provides recourse without a requirement for the 3rd party.

Project and Audit Information

Term	Description
Auditor	Csanuragjain, Minhquanym, Tomo and Ak1
Reviewed by	Raja
Type	ERC-20
Language	Solidity
Ecosystem	Ethereum - Arbitrum
Methods	Manual Review
Repository	https://github.com/unicrowio
Commit hash (at audit start)	919299791f8aaef0f23163f4e5c59eae80010933
Commit hash (after resolution)	ecaefbb74c1d1187abf6ee3ea08baa70a44760f0
Documentation	[Added once the whitepaper is published by the project]
Unit Testing	No
Website	https://unicrow.io/
Submission Time	2022-11-21
Finishing Time	2022-12-14

Contracts in scope

- *contracts/FakeToken.sol*
- *contracts/Unicrow.sol*
- *contracts/UnicrowArbitrator.sol*
- *contracts/UnicrowClaim.sol*
- *contracts/UnicrowDispute.sol*
- *contracts/UnicrowTypes.sol*

Executive Summary

Unicrow's smart contracts were audited between 2022-11-23 and 2022-12-14 by Csanuragjain, Minhquanym, Tomo and Ak1. Manual analysis was carried out on the code base provided by the client The following findings were reported to the client. For more details, refer to the findings section of the report.

S.no.	Issue Category	Issues found	Resolved	Acknowledged
1.	High	7	7	0
2.	Medium	4	2	2
3.	Low	8	5	3
4.	Quality Assurance	6	6	0

Severity definitions

Risk factor matrix	Low	Medium	High
Occasional	L	M	H
Probable	L	M	H
Frequent	M	H	H

High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

Medium: The impact of medium issues is less critical than high, but still probable with considerable damage. The protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions.

Low: Low issues impose a small risk on the project. Although the impact is not estimated to be significant, we recommend fixing them on a long-term horizon. Assets are not at risk: state handling, function incorrect as to spec, issues with comments.

Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

Audit Overview

Code quality: 95.0

Documentation quality: 100.0

Security score: 95.4

Architecture quality: Not in scope

Audit Findings

Finding: #1

Issue: Possibility for owner to withdraw user funds

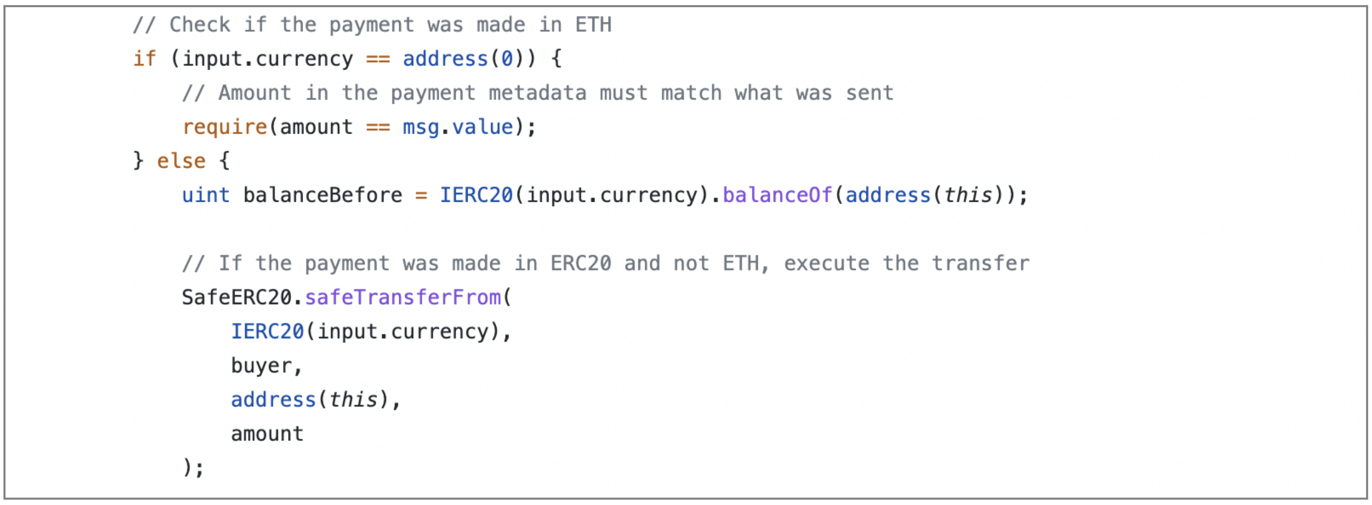
Severity: High

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/Unicrow.sol#L113

Impact: Project owner can withdraw user funds when locked in contract

Description:



```
        // Check if the payment was made in ETH
        if (input.currency == address(0)) {
            // Amount in the payment metadata must match what was sent
            require(amount == msg.value);
        } else {
            uint balanceBefore = IERC20(input.currency).balanceOf(address(this));

            // If the payment was made in ERC20 and not ETH, execute the transfer
            SafeERC20.safeTransferFrom(
                IERC20(input.currency),
                buyer,
                address(this),
                amount
            );
```

1. User calls the pay function with input.currency as `address(0)` and msg.value as `0` and input.amount as `X` and seller as his another wallet address
2. Since msg.value is 0 so below check is skipped
3. Since input.currency is address(0) so no token transfer happens
4. Finally the input amount is stored in the newly created escrow. Remember we have not paid anything till now
5. Now he uses his other wallet address as Seller and calls the refund function.

```
        // Withdraw the amount to the buyer
        if (address(escrow.currency) == address(0)) {
            (bool success, ) = escrow.buyer.call{value: escrow.amount}("");
            require(success, "1-012");
```

6. Since escrow.amount is X, project believes to have received such funds and returns them

Recommendations: Revise the condition like below:

```
if (msg.value != 0) {
// Amount in the payment metadata must match what was sent
  require(input.amount == msg.value);
  require(input.currency == address(0));
} else {
  require(input.currency != address(0));
}
```

Status: Resolved

Finding: #2

Issue: Buyer can withdraw funds intentionally via front-running with proposeArbitrator function

Severity: High

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowArbitrator.sol#L128

Impact: By correctly timing the attack, Buyer can withdraw funds of seller

Description:

1. Escrow was created between Buyer and Seller without any arbitrator

2. Post challenge period, Buyer calls the proposeArbitrator and set some fees (with arbitrator as another address) and immediately calls release function so that escrow could be released to Seller and Arbitrator

3. Even though there was no arbitrator, arbitrator fee set in step2 will be deducted from seller share

Recommendations:

The proposeArbitrator function should not directly update arbitratorData.arbitratorFee variable. Arbitrator could have an additional field like proposedFee which could be updated in such cases and if both party agrees then only proposedFee should become arbitratorFee

Status: Resolved

Finding: #3

Issue: Reentrancy to UnicrowClaim can claim twice

Severity: High

Where:
https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/Unicrow.sol#L223-L232
https://github.com/unicrowio/contracts-audit/blob/main/contracts/Unicrow.sol#L194-L239
https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowClaim.sol#L140-L204

Impact: Attacker can claim an escrow twice, drain all funds in the Unicrow contract

Description: In function \`refund()\`, it set \`claimed = 1\` after withdrawing to buyer. If currency is native token or ERC777 with hook, buyer can take control of the execution flow and call \`UnicrowClaim.singleClaim()\` function. Since \`claimed\` is not set and consensus are set to positive value, it will not revert and allow buyer to claim the same escrow again.

The refund() is used to pay full funds to the buyer by the seller. The singleClaim() is used to pay funds for the buyer,seller, etc ... Both refund() and singleClaim have some
implementation as follows.

1. The nonReentrant modifier to prevent a re-entrancy attack

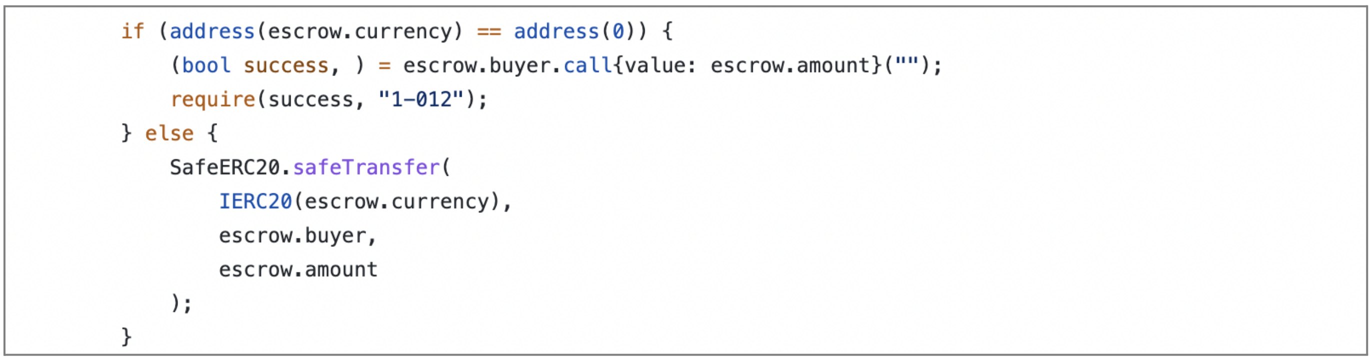
2. The checking of escrow.claimed is 0 to prevent re-claim. However, malicious attackers can claim two times per escrow and all funds in the contract can be stolen.

Example- malicious buyer: Alice, malicious seller: Bob

1. Alice and Bob create an escrow and Alice pay 1 ETH for their orders as input.amount. Their escrow doesn't split for arbitrator and protocol.

2. Bob executes refund()to pay the 1 ETH to Alice as a seller

3. Alice received 1 ETH before the state of claimed changed yet



```
        if (address(escrow.currency) == address(0)) {
            (bool success, ) = escrow.buyer.call{value: escrow.amount}("");
            require(success, "1-012");
        } else {
            SafeERC20.safeTransfer(
                IERC20(escrow.currency),
                escrow.buyer,
                escrow.amount
            );
        }
```

4. (4 not 1) Alice executes singleClaim() as soon as Alice receives ETH in the fallback function

5. (5 not 2) It is true that both refund and singleClaim has the nonReentrant modifier
but these functions are in different contract. Since both contract import "ReentrancyGuard" separately, the state into nonReentrant modifier doesn't synchronize.

6. (6 not 3) Finally, Alice and Bob got 2 ETH from 1 ETH. If we continue this attack, attackers can get all ETH in the contract. Also, if the token follows ERC777, this token will be stolen like ETH.

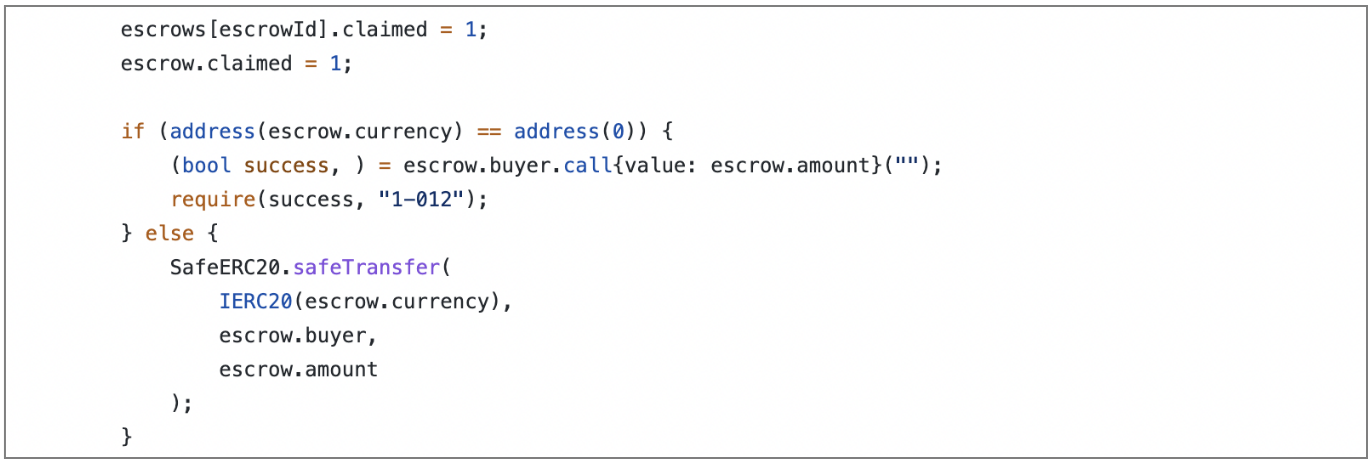
You can see more detail of cross contract reentrancy attack https://inspexco.medium.com/cross-contract-reentrancy-attack-402d27a02a15

Recommendations: Consider following check-effect-interaction pattern or adding nonReentrant to functions `setClaimed()` / `sendEscrowShare()`

"Should change the state of claimed before sending tokens.In other words, you should follow the check-effect-interactions pattern

https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

// Update the escrow as claimed in the storage and in the emitted event



```
        escrows[escrowId].claimed = 1;
        escrow.claimed = 1;

        if (address(escrow.currency) == address(0)) {
            (bool success, ) = escrow.buyer.call{value: escrow.amount}("");
            require(success, "1-012");
        } else {
            SafeERC20.safeTransfer(
                IERC20(escrow.currency),
                escrow.buyer,
                escrow.amount
            );
        }
```

Status: Resolved

Finding: #4

Issue: Any party can reject receiving funds to prevent escrow from settling/claiming

Severity: High

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/Unicrow.sol#L329

https://github.com/unicrowio/contracts-audit/blob/main/contracts/Unicrow.sol#L336

Impact: Escrow cannot be settled.

Any party who is upset about arbitrator split can simply dos the claim functionality. This means all parties involved in escrow will not receive the funds

Description: Since the system adopts ""Push"" pattern to handle funds transferring when settling, if any transfer is failed then the escrow cannot be settled. Any party (buyer, seller, marketplace) can abused this, reject receiving funds (native token or ERC777) and prevent escrow from settling.

1. There was a dispute on escrow for which arbitrator was deciding the split
2. Arbitrator makes below split -> Buyer 80%, Seller 10%, Others 10%
3. Seller is unhappy with Arbitrator decision

4. Once Arbitrator initiates the claim based on his decided split, sendEscrowShare function is called to send each party share
5. Once sendEscrowShare function tries to send split to Seller (who is contract), Seller simply reverts the payment causing success to be false and whole claim to fail

```
    function sendEscrowShare(
        address to,
        uint256 amount,
        address currency
    ) public onlyUnicrowClaim {
         if(currency == address(0)) {
            (bool success, ) = to.call{value: amount}("");
            require(success, "1-012");
         } else {
            SafeERC20.safeTransfer(
                IERC20(currency),
                to,
                amount
            );
         }
     }
```

Recommendations: Consider following Pull over Push pattern when dealing with settling escrow. Do not fail the claim function if party is trying to dos the receive payment. You may store these funds amount in a mapping variable and users can use another function to extract the failed payment using this mapping

Status: Resolved

Finding: #5

Issue: Rounding issue can lead to lacking funds in the contract

Severity: High

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowClaim.sol#L253

Impact: Claiming escrow takes more than expected. If contract has other escrows, they will be lacking funds. If contract has only 1 escrow with that currency, escrow cannot be claimed.

Description: In case `arbitrated = True`, arbitratorFee will be deducted from both buyer and seller splits. However they did not be sum up but after all, in `calculatePayment()`, arbitratorFee is calculated again with total amount.

Scenario (for simplicity, let's just assume there is only arbitrator fee)

1. arbitratorFee = 1000 (10%), split[Buyer] = 4999, split[seller] = 5001

arbitratorFeeFromBuyer = 4999 * 10% = 499, arbitratorFeeFromSeller = 500 sum = 999 != 1000 ??

2. After take fee split[Buyer] = 4500, split[seller] = 4501

3. Total splits = split[Buyer] + split[seller] + arbitratorFee = 10001 > 10000

Recommendations: Consider returning arbitrator with the remaining amount without calculating it using `arbitratorFee`

Status: Resolved

Finding: #6

Issue: Division before multiplication can cause round down issue. Almost in all the places, it is happening

Severity: High

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowArbitrator.sol

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/Unicrow.sol#L383-L385

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowClaim.sol#L263-L277

Impact: value will be round down

Description: Computation is done such that the division first and then multiplication.

Recommendations: Perform multiplication first and then do divide.

Status: Resolved

Finding: #7

Issue: splitCalculation - seller can get unfair amount of share when underflow happen

Severity: High

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/Unicrow.sol#L388-L391

Impact: Seller can get more shares and the percentage will reach maximum limit which could lead to halt the running escrow and offer Settlement will struck

Description: Seller share is calculated inside unchecked state, when underflow happen, seller will receive huge share.

Recommendations: Suggested not use unchecked based share calculation. Enusre, seller received within 100% split

Status: Resolved

Finding: #8

Issue: Rounding down caused leftover funds in the contract

Severity: Medium

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowClaim.sol#L263-L267

Impact: Some token weis are left in the contract. If currency is a low decimal/low cap token, the leftover amount could have high value

Description: In `claimPayment()`, all payments to parties are calculated with rouding down division. It could potentially have some token weis leftover after `claimPayment()`.

Recommendations: Consider returning the last payments the remaining amount without calculating it using rounding division.

Status: Resolved

Finding: #9

Issue: Unsupported for transfer-with-fee token

Severity: Medium

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/Unicrow.sol#L174-L182

Impact: Low amount of funds would be received by vault for transfer with fee tokens

Description: Some ERC20 tokens(e.g. STA, PAXG,in the futureUSDC,USDT), allow for charging a fee any time transfer() or transferFrom() is called.

For more detail, please read this.

https://github.com/d-xo/weird-erc20#fee-on-transfer

Assume that XYZ token is a fee-on-transfer token with a 10% transfer fee. Assume that the user sent 100 XYZ tokens. The accounting system set the amount to 100 but then actual amount of XYZ tokens received by the vault will only be 90 XYZ tokens.

Recommendations: Ensure that to check previous balance/after balance equals to amount for any rebasing/inflation/deflation. Or creating the whitelist to restrict can be used of ERC20 tokens in this protocol.

Status: Resolved

Finding: #10

Issue: Fee could be zero for small decimal tokens

Severity: Medium

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowClaim.sol#L262-L266

Impact: Chance for funds to be stuck in the contract

Description: The fee is calculated as follows payments

```
    [WHO_PROTOCOL] =(uint256(split[WHO_PROTOCOL]) *amount) /_100_PCT_IN_BIPS;
    //_100_PCT_IN_BIPS == 10000
```

It is possible for the calculated fee to be zero for some tokens with small decimals, such as EURS (2 decimals). Therefore, for small-volume transactions, tokens to be paid for protocol or marketplace may be rounded down to zero and stuck in the contract.

Example

```
    split [WHO_PROTOCOL]: 50 (0.5%), amount = 1.8_e2 EURS
    payments [WHO_PROTOCOL] =(uint256(split [WHO_PROTOCOL]) *amount) /_100_PCT_IN_BIPS;
    // 50 * 180 / 10000 = 0
```

Finally, the amount of 50*180 EURS will be stuck in this contract

Recommendations: Such cases are unlikely to occur. However, the possibility of it happening is not zero, so here is a token that can be restricted by the whitelist.

Alternatively, scale the amount to precision (1e4) amounts.

Similar issue

https://github.com/code-423n4/2022-03-sublime-findings/

Status: Acknowledged

Finding: #11

Issue: arbitrator cannot be revoked

Severity: Medium

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowArbitrator.sol

Impact: If contract find that the current arbitrator is misbehaving, it cannot revoke/reset the arbitrator.

Description: There are no functions and way to revoke the arbitrator

Recommendations: Add function to revoke the arbitrator

Status: Acknowledged

Finding: #12

Issue: Two Step Governance Change

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/Unicrow.sol#L286

Impact: All governance function would not be callable if governance is set incorrectly

Description:

1. Observe the updateGovernance function

```
    /// @inheritdoc IUnicrow
    function updateGovernance(address governance) external override onlyGovernance {
        governanceAddress = governance;
    }
```

2. Note that Governance is updated directly.

3. If old governance has provided incorrect address for new governance then governanceAddress is updated incorrectly

Recommendations: Implement 2 step governance change, such that current governance can set a pending governance and then the new governance needs to accept in order to move from pending to confirmed state.

Status: Acknowledged

Finding: #13

Issue: Should change types of decimal to uint8

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowTypes.sol#L127

Impact: NA

Description: The decimal of Token struct used uint256 but usually it is used the type of uint8 for the decimal

```
struct Token {
    address address_;
    uint8 decimals;
    string symbol;
}

/// @dev Superstructure that includes all the information relevant to an escrow
struct Data {
    Escrow escrow;
    Arbitrator arbitrator;
    Settlement settlement;
    Token token;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L87-L89

```
    function decimals() public view virtual override returns (uint8) {
        return 18;
    }
```

Recommendations: You should change as follows.

```
struct Token {
    address address_;
    uint8 decimals;
    string symbol;
}

/// @dev Superstructure that includes all the information relevant to an escrow
struct Data {
    Escrow escrow;
    Arbitrator arbitrator;
    Settlement settlement;
    Token token;
}
```

Status: Resolved

Finding: #14

Issue: address (0) check

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/Unicrow.sol#L286-L288

Impact: NA

Description:

```
    /// @inheritdoc IUnicrow
    function updateGovernance(address governance) external override onlyGovernance {
        governanceAddress = governance;
    }
```

The updateGovernance is used to change the governanceAddress by governance address but if it is executed with address(0) mistakenly, this governance can't change permanently. It is unlikely to happen but you should add this check to prevent such a
mistake.

Recommendations: Update the code as below

```
 function updateGovernance(address governance) external override onlyGovernance {
     require(governance!= address(0) && governanceAddress!= governance);
     governanceAddress = governance;
 }
```

Status: Acknowledged

Finding: #15

Issue: Add immutable keyword

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowClaim.sol#L21

Impact: NA

Description: The comment says to use immutable, but the implementation does not.

```
/// Reference to the main escrow contract (immutable)
    Unicrow public unicrow;
```

Recommendations: You should change as follows.

```
/// Reference to the main escrow contract (immutable)
    Unicrow public immutable unicrow;
```

Status: Resolved

Finding: #16

Issue: arbitrator should not be seller or buyer

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowArbitrator.sol#L109-L145

Impact: arbitrator can either be seller or buyer which will not be help for fair arbitrator

Description: proposeArbitrator is called either by buyer or seller to fix the arbitrator. But it does not check whether the arbtrator is buyer or seller

Recommendations: Ensure that the arbitrator should not be either buyer or seller

Status: Acknowledged

Finding: #17

Issue: Marketplace can be zero address but still having `marketplaceFee` makes it impossible to claim the escrow

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/Unicrow.sol#L125

Impact: Escrow cannot be claimed/settled. Escrow funds are locked in contract

Description: Lacking zero address check for `marketplace` address in function `pay()`. `marketplaceFee` can be non-zero value but `marketplace` is zero address. During claiming, transferring to zero address will fail and prevent escrow from claimed

Recommendations: Consider adding zero address check for `marketplace` in case `marketplaceFee > 0`

Status: Resolved

Finding: #18

Issue: Buyer can mistakenly send ETH to contract and ETH is locked

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/Unicrow.sol#L137

Impact: ETH mistakenly send through `pay()` is not returned to buyer but locked in contract.

Description: In case `currency` is not ETH but buyer mistakenly send ETH in `pay()` function, transaction is not reverted. It still transfer ETH and currency token to the contract. Buyer cannot redeem or claim these ETH back.

Recommendations: Consider adding check for `currency` in case `msg.value > 0`

Status: Resolved

Finding: #19

Issue: No handling with msg.value but there is a payable keyword

Severity: Low

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowClaim.sol#L82

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowClaim.sol#L140

Impact: NA

Description: Both singleClaim() and claim() has payable keyword but there is no handling of msg.value. This means that this function can receive ETH as a msg.value but this msg.value doesn't use.

Finally, ETH sent to this function as msg.value will be stuck in this contract. Also, if the claim () has a handling for msg.value, it has the risk of msg.value inside a loop

https://github.com/crytic/slither/wiki/Detector-Documentation#msgvalue-inside-a-loop

Recommendations: Delete the payable keyword.

Status: Resolved

Finding: #20

Issue: Nothing is allowed at escrow.challengePeriodEnd

Severity: Quality Assurance

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowDispute.sol#L89

Impact: Nothing can actually be done at exact escrow.challengePeriodEnd

Description:

1. You are not allowed to challenge at exact escrow.challengePeriodEnd as shown below

```
function challenge(uint256 escrowId) external override nonReentrant {
        ...

        // Check that the challenge period is running
        require(block.timestamp < escrow.challengePeriodEnd, "1-016");

        ...
}
```

2. Also you are not allowed to release the payment at exact escrow.challengePeriodEnd (Note release function calls singleClaim function which have below check)

```
 require(
        (escrow.consensus[WHO_SELLER] >= 1 &&
           escrow.consensus[WHO_BUYER] >= 1) ||
           block.timestamp > escrow.challengePeriodEnd,
         ""0-006""
        );
```

3. So nothing can actually be done at exact escrow.challengePeriodEnd

Recommendations: Update the challenge period till escrow.challengePeriodEnd

```
function challenge(uint256 escrowId) external override nonReentrant {
        ...

        // Check that the challenge period is running
        require(block.timestamp <= escrow.challengePeriodEnd, "1-016");

        ...
}
```

Status: Resolved

Finding: #21

Issue: Optimize number of external calls

Severity: Quality Assurance

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowArbitrator.sol#L74

Impact: NA

Description: It is noticed that contracts (Unicrow, UnicrowClaim, UnicrowDispute,...) communicate, call each other frequently. Even some view function are called again and again. It is really gas consuming and not recommended.

For example, in `onlyEscrowMember` modifier, it does 2 external call to the same function `unicrow.getEscrow()` while it can be optimized easily with only 1 external call

Recommendations: Consider reviewing the external calls again and optimizing the gas cost for users.

Status: Resolved

Finding: #22

Issue: Caching number of storage load to save gas

Severity: Quality Assurance

Where:

https://github.com/unicrowio/contracts-audit/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowDispute.sol#L190

Impact: NA

Description: Accessing variable from memory/stack is much cheaper than from storage, it is recommended to cache the value loaded from storage when it is used multiple times.

For example, in function `approveSettlement()`, `latestSettlementOfferBy[escrowId]` is loaded from storage 2 times

Recommendations: Consider caching the value loaded from storage to save gas

Status: Resolved

Finding: #23

Issue: Use fixed value not array.length

Severity: Quality Assurance

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowClaim.sol#L297

Impact: NA

Description: The length of amounts is always 5 but the length is fetched by each loop

```
function claimPayments(
    uint escrowId,
    uint[5] memory amounts,
    address[5] memory addresses,
    address currency
) internal {
    unicrow.setClaimed(escrowId);

    for (uint256 i = 0; i < amounts.length; ++i) {
        if (amounts[i] > 0) {
            unicrow.sendEscrowShare(addresses[i], amounts[i], currency);
        }
    }
}
```

Recommendations: Update the function as follows

```
function claimPayments(
    uint escrowId,
    uint[5] memory amounts,
    address[5] memory addresses,
    address currency
) internal {
    unicrow.setClaimed(escrowId);

    for (uint256 i = 0; i < 5; ++i) {
        if (amounts[i] > 0) {
            unicrow.sendEscrowShare(addresses[i], amounts[i], currency);
        }
    }
}
```

Status: Resolved

Finding: #24

Issue: Use external instead of public

Severity: Quality Assurance

Where:

https://github.com/unicrowio/contracts-audit/blob/main/contracts/UnicrowDispute.sol#L228-L230

Impact: NA

Description:

```
function getSettlementDetails(uint256 escrowId) public view returns (Settlement memory) {
        Settlement memory settlement = Settlement(latestSettlementOfferBy[escrowId], latestSettlementOffer[escrowId]);
        return settlement;
    }
```

The visibility of getSettlementDetails is public but this function doesn't call in the UnicrowDispute contract. Therefore, this visibility should change from public to external in terms of gas-optimization

You can see for more details :

https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices

Recommendations: Update the function as follows

```
function getSettlementDetails(uint256 escrowId) external view returns (Settlement memory) {
        Settlement memory settlement = Settlement(latestSettlementOfferBy[escrowId], latestSettlementOffer[escrowId]);
        return settlement;
    }
```

Status: Resolved

Finding: #25

Issue: Usage of post increment is not encouraged

Severity: Quality Assurance

Where:

https://github.com/unicrowio/contracts/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowClaim.sol#L86

https://github.com/unicrowio/contracts/blob/919299791f8aaef0f23163f4e5c59eae80010933/contracts/UnicrowClaim.sol#L297

Impact: could cost more gas

Description: using post increment in loops could cause more gas

Recommendations: Use pre-increment

Status: Resolved

Disclaimer