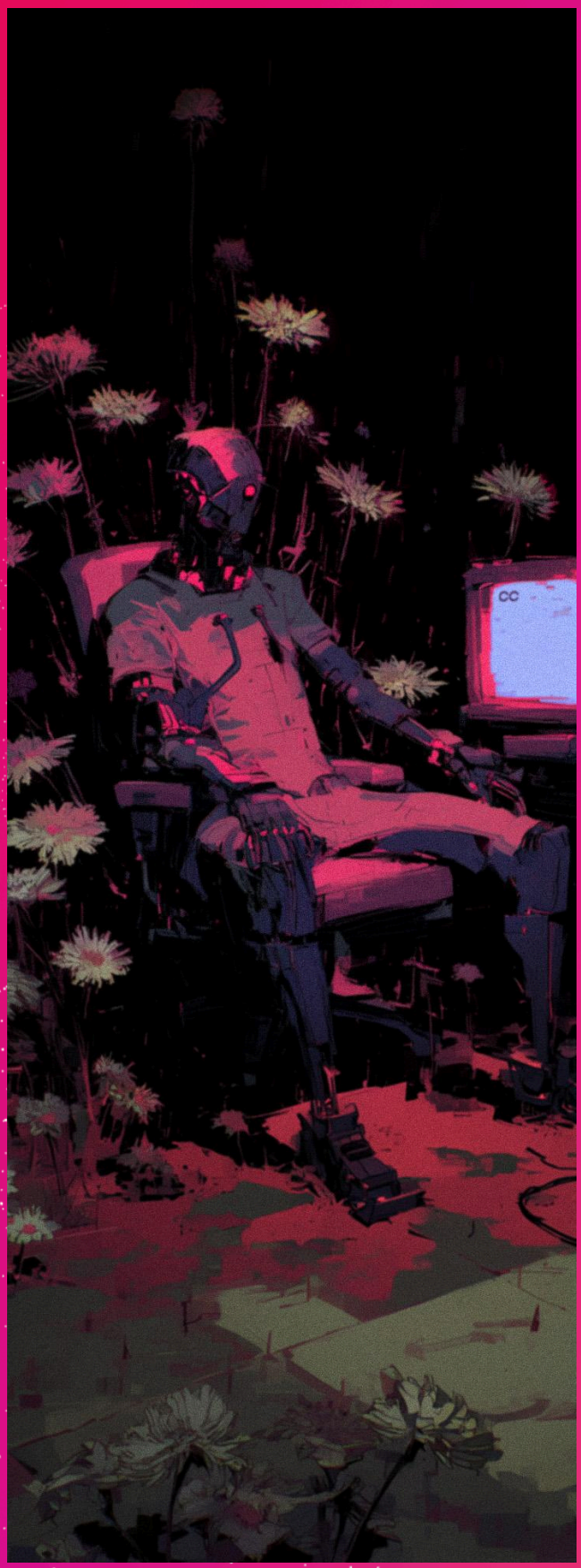
IMMUNEFI AUDIT

Immunefi / CC Protocol



DATE August 5, 2025

AUDITOR Neplox, Security Researchers

REPORT BY Immunefi

ABOUT IMMUNEFI

Immunefi is the leading onchain security platform, having directly prevented hacks worth more than $25 billion USD. Immunefi security researchers have earned over $120M USD for responsibly disclosing over 4,000 web2 and web3 vulnerabilities, more than the rest of the industry combined.

Through Magnus, Immunefi delivers a comprehensive suite of best-in-class security services through a single command center to more than 300 projects — including Sky (formerly MakerDAO), Optimism, Polygon, GMX, Reserve, Chainlink, TheGraph, Gnosis Chain, Lido, LayerZero, Arbitrum, StarkNet, EigenLayer, AAVE, ZKsync, Morpho, Ethena, USDT0, Stacks, Babylon, Fuel, Sei, Scroll, XION, Wormhole, Firedancer, Jito, Pyth, Eclipse, PancakeSwap and many more.

Magnus unifies SecOps across the entire onchain lifecycle, combining Immunefi's market leading products and community of elite security researchers with a curated set of the very best security products and technologies provided by top security firms — including Runtime Verification, Dedaub, Fuzzland, Nexus Mutual, Failsafe, OtterSec and others.

Magnus is powered by Immunefi's proprietary vulnerabilities dataset — the largest and most comprehensive in web3, ensuring that security leaders and teams have the best possible tools for identifying and mitigating life threats before they cause catastrophic harm, all while reducing operational overhead and complexity.

Learn how you can benefit too at immunefi.com.

TERMINOLOGY

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

LIKELIHOOD	IMPACT		
	HIGH	MEDIUM	LOW
CRITICAL	Critical	Critical	High
HIGH	High	High	Medium
MEDIUM	Medium	Medium	Low
LOW	Low		
NONE	None		

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

EXECUTIVE SUMMARY

Over the course of 7 days in total, CC Protocol engaged with Immunefi to review the ccprotocol-contracts-internal repository. In this period of time a total of 21 issues were identified.

SUMMARY

Name	CC Protocol
Repository	https://github.com/ccprotocol/ccprotocol-contracts-internal
Audit Commit	e44a2d34429de9ba8f5fc9a984ee600dada6289b
Type of Project	Infrastructure
Audit Timeline	July 1st - July 7th
Fix Period	July 10th - Aug 4th

ISSUES FOUND

Severity	Count	Fixed	Acknowledged
Critical	1	1	0
High	3	2	1
Medium	5	3	2
Low	6	2	4
Insights	6	6	0

CATEGORY BREAKDOWN

Bug	15
Gas Optimization	1
Informational	5

FINDINGS

IMM-CRIT-01

`KeepWhatsRaised` withdrawals can be manipulated to leave no funds for campaign owner #15

Id	IMM-CRIT-01
Severity	Critical
Category	Bug
Status	Fixed in e8ccf17c96143d0e8c401d27c8d61175b2782419

Description

The `KeepWhatsRaised` treasury implements a complex withdrawal scheme in its `withdraw` method with varying post-/pre-deadline withdrawal logic, varying schemes, and Columbian tax accounting. Withdrawals are open as soon as the platform admin enables them through `approveWithdrawal`, and are available up until the withdrawal delay after the campaign deadline (enforced through the `currentTimeIsLess(getDeadline() + s_config.withdrawalDelay)` modifier call in the method). Pre-deadline withdrawals require specifying the amount to withdraw, which will be deducted from the pledged amount storage variable `s_availablePledgedAmount`. The withdrawal amount is split into fees which go to the platform, protocol fees, and the value which actually goes to the campaign owner.

An interesting functionality of the `withdraw` method of the treasuries (both `AllOrNothing` and `KeepWhatsRaised`) we have noted is the lack of access control enforced: any user or contract on the network where the contracts are deployed is able to perform the withdrawal. Supposedly, this was planned to be justified by the withdrawal implementation, which always transfers funds to the campaign owner and nobody else.

In the case of `AllOrNothing`, we, indeed, do not consider there to be any issue with the `withdraw` method being callable by anyone. `KeepWhatsRaised`'s case, however, is more complex, since it performs the fee accounting logic within `withdraw`. We'd like to point out one specific kind of fees accounted for in `KeepWhatsRaised.withdraw`: flat rate platform fees, specified by the `s_feeKeys.flatFeeKey` and

`s_feeKeys.cumulativeFlatFeeKey` platform data values. These fees are subtracted from the withdrawalAmount without being scaled or transformed in any manner:

```
TypeScript
function withdraw(
    uint256 amount
)
    public
    currentTimeIsLess(getDeadline() + s_config.withdrawalDelay)
    whenNotPaused
    whenNotCancelled
    withdrawalEnabled
{
    ...

    // Flat fee calculation
    if(currentTime > getDeadline()){
        if(withdrawalAmount == 0){
            revert KeepWhatsRaisedAlreadyWithdrawn();
        }
        if(withdrawalAmount < s_config.minimumWithdrawalForFeeExemption){
                s_platformFee += flatFee;
                totalFee += flatFee;
        }
    } else {
        withdrawalAmount = amount;
        if(withdrawalAmount == 0){
            revert KeepWhatsRaisedInvalidInput();
        }
        if(withdrawalAmount > s_availablePledgedAmount){
            revert KeepWhatsRaisedWithdrawalOverload(s_availablePledgedAmount,
withdrawalAmount, totalFee);
        }

        if(withdrawalAmount < s_config.minimumWithdrawalForFeeExemption){
                s_platformFee += cumulativeFee;
                totalFee += cumulativeFee;
        } else {
            s_platformFee += flatFee;
            totalFee += flatFee;
        }
    }

    // Other Fees
```

```
    ...

    // Fees subtracted from withdrawal amount
    s_availablePledgedAmount -= withdrawalAmount;
    withdrawalAmount -= totalFee;

    TOKEN.safeTransfer(recipient, withdrawalAmount);

    emit WithdrawalWithFeeSuccessful(recipient, withdrawalAmount, totalFee);
}
```

Note how these fees are subtracted during each `withdraw` call, and there is no limit on the minimal withdrawal amount, meaning that the resulting `withdrawalAmount` might even be zero after accounting for the flat rate fees. By combining this with the lack of access controls on the `withdraw` method, we have come to the realization that a malicious actor can repeatedly call `withdraw` with tiny `amount` values, just enough to cover the flat fees, which would account all of the pledged funds as platform fees instead of funds transferred to the owner. By monitoring `Receipt` events, the attacker can perform these withdrawals immediately (granted that they were enabled by the platform admin using `approveWithdrawal`), leaving the owner without the chance to receive any funds at all.

Furthermore, this issue can be exploited effortlessly in all instances of the `KeepWhatsRaised` treasury, in all campaigns launched on the CC Protocol using it. Such an attack would completely break the protocol, as none of the campaign owners would actually be left with any funds after their campaign finishes. Resolving the issue would require the platforms and the protocol to collaborate and manually recalculate all the pledges made through `KeepWhatsRaised` in order to correctly distribute them. With all of these points in mind, we consider the issue to be of critical severity.

Recommendation

Rethink the withdrawal logic, perhaps limiting access to the `withdraw` method to only the owner of the campaign and the platform admin (for automated withdrawals). Reconsider the fee structure in `KeepWhatsRaised` treasury to avoid flat fees, if possible, or at least enforce minimal withdrawal amounts, so that the withdrawal amount is not zero after flat fee deduction.

IMM-HIGH-01

KeepWhatsRaised updateDeadline can block claimRefund by setting past deadline #5

Id	IMM-HIGH-01
Severity	High
Category	Bug
Status	Fixed in 8ac139306900a9df6c73b4455b0607415cbe4773

Description

The `updateDeadline` function in `KeepWhatsRaised` contract contains a vulnerability that allows campaign owners to instantly block user refunds by setting the campaign deadline to a past timestamp.

The `updateDeadline` function only validates that the new deadline is greater than the launch time, but does not require the deadline to be in the future:

```
TypeScript
function updateDeadline(
    uint256 deadline
) external
    onlyPlatformAdminOrCampaignOwner
    onlyBeforeConfigLock
    whenNotPaused
    whenNotCancelled
{
    if (deadline <= getLaunchTime()) {  // Only checks against launch time
        revert KeepWhatsRaisedInvalidInput();
    }
    // Missing: deadline > block.timestamp check
    s_campaignData.deadline = deadline;
}
```

The `claimRefund` function depends on the deadline through `_checkRefundPeriodStatus`:

```TypeScript
function claimRefund(uint256 tokenId) external {
    if (!_checkRefundPeriodStatus(false)) {
        revert KeepWhatsRaisedNotClaimable(tokenId);
    }
    // refund logic...
}

function _checkRefundPeriodStatus(bool checkIfOver) internal view returns (bool) {
    uint256 deadline = getDeadline();
    bool isCancelled = s_cancellationTime > 0;

    if (!isCancelled) {
        // For non-cancelled campaigns: refund possible only AFTER deadline but BEFORE refund
delay expires
        return block.timestamp > deadline && !refundPeriodOver;
    }
}
```

1. Campaign is running normally with deadline in the future
2. Campaign owner sets deadline to N hour ago
3. If `refundDelay` period has already passed since that past time, refunds become impossible forever

Recommendation

Add validation to ensure deadline is always in the future:

```TypeScript
if (deadline <= block.timestamp) {
    revert KeepWhatsRaisedInvalidInput();
}
```

IMM-HIGH-02

Griefing and freezing of funds through unauthorized pledge calls #7

Id	IMM-HIGH-02
Severity	High
Category	Bug
Status	Acknowledged Fix ready but not released in 14dd9e990605d740bb20f13d1468b78fab020f47

Description

As described in issue 3.2 of the CC Protocol audit by PeckShield, the treasuries' `pledge` methods (`pledgeForAReward`, `pledgeWithoutAReward`) allow unauthorized pledging on behalf of backers who have `approve()`d the protocol token spending by a campaign treasury. From the resolution comment in the report, we understand that this issue has been acknowledged by the team, and the pledging feature implementation is intended to be this way. However, PeckShield failed to list the appropriate impacts caused by this issue, which we feel should make it worthwhile to rework the flow.

Since a malicious user (absolutely any on-chain address) can call pledge on behalf of a backer who has approved token spending, they can block the backer's legitimate `pledgeForAReward` or `pledgeWithoutAReward` calls from succeeding, causing the backer (or the system pledging on behalf of the backer) to lose funds on gas payments. Token spending approvals are easily trackable thanks to event logs emitted by pretty much all ERC20 implementations, so an attacker could monitor approvals for the token used by the protocol to treasury contracts, and instantly execute pledge calls (for example, `pledgeWithoutAReward`) with tiny sums, just enough for the call to succeed. Legitimate pledge requests would fail due to insufficient funds being left in the backer's approval. The backer would have to complete a refund, and only then retry the pledge. A persistent attacker could use this issue to completely block pledges to a treasury, sabotaging the campaign it is linked to.

Both `AllOrNothing` and `KeepWhatsRaised` treasuries are affected. The current implementation of KeepWhatsRaised is affected even more so, as it allows refunds to go through only after cancellation or after the deadline (see `KeepWhatsRaised._checkRefundPeriodStatus`). This attack would effectively lock the backer's funds in the campaign without the ability to choose the rewards they want. We consider this to be an issue far more severe than was initially described by PeckShield in their report. Since in the

`KeepWhatsRaised` treasury this issue can be exploitable to temporarily freeze the funds of the backers due to the refund mechanism, we are rating the vulnerability severity as High.

Recommendation

If indirect pledging is required, for example gas-free pledge transactions, well-established mechanisms should be used instead, e.g. EIP-2612 or UniSwap's Permit2. If this behaviour and vulnerability is indeed intentional and there are some more complex reasons why it can't be fixed, the documentation for all the `pledge` methods should mention that they should be called from contracts which execute the `approve` and `pledge` in a single transaction.

IMM-HIGH-03

Backers can pledge for rewards without actually paying for them #14

Id	IMM-HIGH-03
Severity	High
Category	Bug
Status	Fixed in `bf841bdb0fb9113ba8a93d1e88d2f7c4b44eb4b2`

Description

Both of the existing treasury protocol implementations, `AllOrNothing` and `KeepWhatsRaised`, allow backers to pledge both for rewards and without them using the `pledgeForAReward` and `pledgeWithoutAReward` methods, accordingly.

Both `AllOrNothing.pledgeForAReward` and `KeepWhatsRaised.pledgeForAReward` function pretty similarly: the backer specifies the list of rewards they'd like to pledge for, the method aggregates the prices for those rewards, which are set by the campaign owner through the treasury's `addRewards` method, and passes the sum as the `pledgeAmount` to the treasury-internal `_pledge` method. `_pledge` transfers the total sum of the rewards and other pledged funds from the backer.

Example from the `KeepWhatsRaised` treasury:

```
TypeScript
function pledgeForAReward(
    bytes32 pledgeId,
    address backer,
    uint256 tip,
    bytes32[] calldata reward
)
    public
    currentTimeIsWithinRange(getLaunchTime(), getDeadline())
    whenCampaignNotPaused
    whenNotPaused
    whenCampaignNotCancelled
    whenNotCancelled
{
```

```
    ...
    uint256 rewardLen = reward.length;
    Reward memory tempReward = s_reward[reward[0]];
    ...
    uint256 pledgeAmount = tempReward.rewardValue;
    for (uint256 i = 1; i < rewardLen; i++) {
        if (reward[i] == ZERO_BYTES) {
            revert KeepWhatsRaisedInvalidInput();
        }
        pledgeAmount += s_reward[reward[i]].rewardValue;
    }
    _pledge(pledgeId, backer, reward[0], pledgeAmount, tip, tokenId, reward);
}

function _pledge(
    bytes32 pledgeId,
    address backer,
    bytes32 reward,
    uint256 pledgeAmount,
    uint256 tip,
    uint256 tokenId,
    bytes32[] memory rewards
) private {
    uint256 totalAmount = pledgeAmount + tip;
    TOKEN.safeTransferFrom(backer, address(this), totalAmount);
    ...
}
```

What we have noticed is that neither `AllOrNothing.pledgeForAReward` nor `KeepWhatsRaised.pledgeForAReward` validate any elements of the `reward` argument to specify actually available rewards besides the first one, which is checked to have the `isRewardTier` flag set:

TypeScript

```
Reward storage tempReward = s_reward[reward[0]];
if (
    backer == address(0) ||
    rewardLen > s_rewardCounter.current() ||
    reward[0] == ZERO_BYTES ||
    !tempReward.isRewardTier // <-- the only check related to the reward list parameter
```

```
) {
    revert AllOrNothingInvalidInput();
}
```

In normal use, this shouldn't be an issue: the owner can add and remove rewards dynamically (using the `addRewards` and `removeReward` methods), so rewards pledged for aren't supposed to be valid during the whole duration of the campaign. However, the lack of this validation leads can be exploited by an attacker to pledge for real rewards before they are registered in the campaign treasury, either by using well-known reward identifiers (if such can exist), retrieving them from external sources (e.g. the API of the platform which hosts the campaign), or by front-running all `addRewards` calls made by crowdfunding owners across the whole CC Protocol. If pledging for rewards which do not yet exist at the moment of the pledge, they will be contained in the `Receipt` event emitted by the `_pledge` methods of the treasuries, but will not be accounted for in the pledge, effectively costing zero. The attacker would only have to pay for gas fees and protocol/platform fees (if any), which in any normal situation should be much less than the rewards themselves would cost.

Considering that both the `AllOrNothing` and `KeepWhatsRaised` treasuries are susceptible to this issue, and its exploitation does not have any unusual preconditions, we are rating it as being of high severity.

Recommendation

As remediation we recommend adding extra validation steps to both `AllOrNothing.pledgeForAReward` and `KeepWhatsRaised.pledgeForAReward` which check that the `reward` parameter contains rewards actually registered in the treasury.

Checking the `s_reward[reward[i]].rewardValue != 0` condition should be valid, since the `addRewards` methods specifically check that added rewards' values are non-zero:

```
TypeScript
function addRewards(
    bytes32[] calldata rewardNames,
    Reward[] calldata rewards
)
    external
    onlyCampaignOwner
    whenCampaignNotPaused
    whenNotPaused
```

```
    whenCampaignNotCancelled
    whenNotCancelled
{
    if (rewardNames.length != rewards.length) {
        revert KeepWhatsRaisedInvalidInput();
    }

    for (uint256 i = 0; i < rewardNames.length; i++) {
        bytes32 rewardName = rewardNames[i];
        Reward calldata reward = rewards[i];

        // Reward name must not be zero bytes and reward value must be non-zero
        if (rewardName == ZERO_BYTES || reward.rewardValue == 0) {
            revert KeepWhatsRaisedInvalidInput();
        }
        ...
    }
    ...
}
```

IMM-MED-01

Blocking `CampaignInfoFactory.createCampaign` calls via front-running #1

Id	IMM-MED-01
Severity	Medium
Category	Bug
Status	Acknowledged Fix ready but not released in fec67cfe084c26d4a08f4b0444ac08a9247a760e

Description

`CampaignInfoFactory` is meant to be the dedicated entrypoint of the CC Protocol for creating crowdfunding campaigns across various platforms. However, the implementation of its factory method, `createCampaign`, allows an attacker to actively block the creation of new `CampaignInfo` contracts, effectively bricking the protocol until the attack is stopped. The attack is not economically expensive, as it requires only payments for gas to be made without additional token or coin transfers.

`CampaignInfoFactory.createCampaign` takes an `identifierHash` parameter which it uses as a unique ID for the campaign, besides the `clone` address itself, of course. This ID isn't concatenated/combined with any other identifying information (e.g. `msg.sender`), which allows an attacker to grief the `CampaignInfoFactory` contract by front-running valid calls to `CampaignInfoFactory.createCampaign` and creating campaigns using the same `identifierHash` as specified in the valid calls.

`createCampaign` checks whether a campaign exists using the given identifier in the validation stage of the method and reverts if a conflict is identified:

```
TypeScript
function createCampaign(
    address creator,
    bytes32 identifierHash,
    bytes32[] calldata selectedPlatformHash,
    bytes32[] calldata platformDataKey,
    bytes32[] calldata platformDataValue,
    CampaignData calldata campaignData
```

```
) external override {
    ...
    address cloneExists = identifierToCampaignInfo[identifierHash];
    if (cloneExists != address(0)) {
        revert CampaignInfoFactoryCampaignWithSameIdentifierExists(
            identifierHash,
            cloneExists
        );
    }
```

Since there are no upfront costs for creating a campaign, a malicious counterparty can effectively block all calls to `createCampaign`. Additionally, depending on how `CampaignInfo` contracts are linked to the crowdfunding campaigns located on off-chain platforms, other user's crowdfunding campaigns might be registered with the attacker as the owner: since the `createCampaign` call can be front-run, an attacker can supply the same `identifierHash` used for identifying the campaign on off-chain platforms, but specify their own address as the `owner`.

Recommendation

Use identifiers validated to be linked to a campaign `creator`, for example, by hashing the concatenation of `creator` and `identifierHash` values. On top of this, security can be improved further by receiving and validating a signature made by the creator for the hash of the concatenation of `creator` and `identifierHash`. Receiving a signature of the `identifierHash` value itself would be insufficient as an attacker would be able to sign it using their own key and supply their own address as the `creator`.

IMM-MED-02

Blocking `KeepWhatsRaised` pledge functions via front-running #4

Id	IMM-MED-02
Severity	Medium
Category	Bug
Status	Fixed in 1d6ad873f7ca30cd6eab33cfff39022508149dad

Description

The `KeepWhatsRaised` contract contains a critical design flaw in its pledge functions (`pledgeWithoutAReward` and `pledgeForAReward`) that allows attackers to perform front-running attacks against specific campaigns.

Both pledge functions accept a pledgeId parameter that must be unique:

```
TypeScript
    /**
     * @notice Allows a backer to pledge for a reward.
     * @dev The first element of the `reward` array must be a reward tier and the other
elements can be either reward tiers or non-reward tiers.
     *      The non-reward tiers cannot be pledged for without a reward.
     * @param pledgeId The unique identifier of the pledge.
     * @param backer The address of the backer making the pledge.
     * @param tip An optional tip can be added during the process.
     * @param reward An array of reward names.
     */
    function pledgeForAReward(
        bytes32 pledgeId,
        address backer,
        uint256 tip,
        bytes32[] calldata reward
    )
```

If a pledgeId is already used, the transaction reverts:

```typescript
if(s_processedPledges[pledgeId]){
    revert KeepWhatsRaisedPledgeAlreadyProcessed(pledgeId);
}
s_processedPledges[pledgeId] = true;
```

This ID isn't concatenated/combined with any other identifying information (e.g. `msg.sender`), which allows an attacker to grief the `KeepWhatsRaised` contract by front-running valid calls to pledge functions using the same `pledgeId`.

Attackers can monitor blockchain for pending pledge transactions, extract the `pledgeId` values, and front-run them with zero-amount pledges using the same IDs. After this, the user's request with that `pledgeId` will revert.

This allows attackers to effectively block all calls to `pledgeWithoutAReward` and `pledgeForAReward`.

Recommendation

Replace externally-provided pledge IDs with internally generated ones. It's sufficient to use `msg.sender` in the `pledgeId` generation.

IMM-MED-03

`AllOrNothing` treasury ends up with locked funds if canceled after success condition #8

Id	IMM-MED-03
Severity	Medium
Category	Bug
Status	Acknowledged Fix ready but not released in ad31321321584bbdf81e39c5df94ce88c2e0a154

Description

For legal purposes and to protect the backers malicious or otherwise non-compliant campaign owners, the platform admin is able to pause and cancel the treasury contracts used by campaigns on their platform. Additionally, the owner themselves is able to cancel specific treasuries (e.g. `KeepWhatsRaised.cancelTreasury` supports cancellation by both owner and platform admin), or the campaign.

The global protocol admin is also able to cancel campaigns (`CampaignInfo._cancelCampaign`). Overall, the CC Protocol implements a pretty advanced and layered pausing/cancelation scheme here, all for the sake of the backers.

One of the core requirements for cancelation is that it must not block backer refunds, so that backers are able to withdraw their pledged funds from a canceled campaign. See `AllOrNothing.claimRefund`, for example: it is blocked when a treasury is paused (indicated a temporarily frozen state enabled to investigate or verify the campaign), but does depend on the cancelation of the treasury:

```
TypeScript
function claimRefund(
    uint256 tokenId
)
    external
    currentTimeIsGreater(INFO.getLaunchTime())
    whenCampaignNotPaused // <-- checks only for campaign/treasury pause
```

```
    whenNotPaused
{
    if (block.timestamp >= INFO.getDeadline() && _checkSuccessCondition()) {
        revert AllOrNothingNotClaimable(tokenId);
    }
    ...
}
```

`KeepWhatsRaised.claimRefund` functions in a similar way, but additionally checks for cancellation to allow refunds for a set time after cancellation. `AllOrNothing`, however, has a subtle issue in the `claimRefund` method, which prevents refunds from succeeding if the campaign or treasury is canceled when the success condition is passed: the very beginning of the `claimRefund` method listed above contains a check that prevents refunds after the deadline and when `_checkSuccessCondition` is true (which happens when enough funds are raised). Even if the treasury/campaign is canceled at this point, the refund won't work, completely breaking the core requirement of cancelations mentioned earlier.

Furthermore, since the `disburseFees` and `withdraw` methods of `AllOrNothing` will be blocked, too (as they should be, since cancelation means that no funds should be distributed to the campaign owner or platform), there will be no way to unlock the funds (cancelation is a one-way process, it isn't possible to "un-cancel" the treasury or campaign). They will remain frozen forever on the treasury contract.

An additional issue worth to be highlighted here is that the cancellation can occur between calls to `disburseFees` and withdraw, which would leave the contract in a state which would make no sense even if the issue with refunds isn't taken into account. Since fees have already been disbursed, some of the pledged funds have been transferred. As such, if backers will attempt to refund their pledges, some of them will not be successful due to insufficient funds being left in the treasury.

Recommendation

Cancelations must be allowed even after the campaign goal is met — this was confirmed with the CC Protocol team during a meeting.

As such, the `claimRefund` of `AllOrNothing` must be adjusted to account for cancellations after the deadline has passed and the goal has been reached. Additionally, cancellations must not be allowed to pass between disburseFees and withdraw calls as they will lead to protocol insolvency: despite the treasury/campaign being canceled, not all backers will be able to refund their pledges.

This might require re-organizing the `disburseFees` and `withdraw` methods into a single method which calculates the fees and payouts and stores them in the contract state, for all the parties to be able to withdraw through separate calls. This way, there'd be no way for a cancellation to occur in between these two method calls.

IMM-MED-04

Gateway fee bypass #19

Id	IMM-MED-04
Severity	Medium
Category	Bug
Status	Fixed in 783fb90a05982c899ba0fdcd92d7d3ce975e2d14

Description

Backers can completely bypass gateway fees by calling `pledgeForAReward()` or `pledgeWithoutAReward()` directly instead of using the intended `setFeeAndPledge()` function. Gateway fees are only applied when they exist in the `s_paymentGatewayFees` mapping, which is not populated during direct pledge calls.

Gateway fees are calculated in `_calculateNetAvailable()` by looking up the fee amount using `getPaymentGatewayFee(pledgeId)`. However, if no fee was previously set for a pledgeId, this function returns 0, effectively bypassing all gateway fees.

Moreover, it's impossible to set a payment gateway fee later through setPaymentGatewayFee because the fee will already be calculated in `_calculateNetAvailable` when executing `pledgeForAReward()/pledgeWithoutAReward()`.

TypeScript

```
function _calculateNetAvailable(bytes32 pledgeId, uint256 tokenId, uint256 pledgeAmount)
internal returns (uint256) {
    uint256 totalFee = 0;

    // ... other fee calculations

    //Payment Gateway Fee Calculation
    uint256 paymentGatewayFee = getPaymentGatewayFee(pledgeId);  // Returns 0 if not set
    s_platformFee += paymentGatewayFee;
    totalFee += paymentGatewayFee;

    return pledgeAmount - totalFee;
}
```

Given that they always deduct money from the backer's wallet, this completely eliminates the purpose of `setFeeAndPledge`. Because it's always more profitable for the backer to directly use `pledgeForAReward()`/`pledgeWithoutAReward()`.

Recommendation

We recommend reconsidering the logic related to gateway fees. Perhaps it's worth calculating payment gateway fees globally or adding fee recalculation logic.

With both `setFeeAndPledge` and `pledgeForAReward()`/`pledgeWithoutAReward()`, the backer needs to do an onchain approval. It's unclear why this logic exists if onchain transactions need to be done anyway. It would make sense if the `transferFrom` came from msg.sender instead of the backer. Because then the payment `gatewayFee` would be paid when the admin pays for the backer from their wallet

IMM-MED-05

`configureTreasury` leads to restrictions bypass #20

Id	IMM-MED-05
Severity	Medium
Category	Bug
Status	Fixed in 4015791a42ad401fcf65757f509cce342ae95c5b

Description

Platform admin can call `configureTreasury`, which overrides all company data including deadline and `goalAmount`, without any checks on updated data.

```TypeScript
    function configureTreasury(
        Config memory config,
        CampaignData memory campaignData,
        FeeKeys memory feeKeys
    )
        external
        onlyPlatformAdmin(PLATFORM_HASH)
        whenCampaignNotPaused
        whenNotPaused
        whenCampaignNotCancelled
        whenNotCancelled
    {
        s_config = config;
        s_feeKeys = feeKeys;
        s_campaignData = campaignData;

        emit TreasuryConfigured(
            config,
            campaignData,
            feeKeys
        );
    }
```

The problem is that this completely bypasses all logic with `deadlineLimit` in the `claimFund` function

```TypeScript
uint256 deadlineLimit = getDeadline() + s_config.withdrawalDelay;
```

In fact, platform admin can at any moment set deadlines in the past and withdraw all money through `claimFund`, bypassing existing restrictions.

Recommendation

Add checks inside `configureTreasury` based on `block.timestamp` for deadline. Also, in general, it's worth considering moving away from an architecture where platform admin can change any campaign.

IMM-LOW-01

`GlobalParams.delistPlatform` can break existing campaigns #2

Id	IMM-LOW-01
Severity	LOW
Category	Bug
Status	Acknowledged

Description

`GlobalParams` is a singleton configuration contract used across pretty much all CC Protocol contracts to retrieve shared parameters controlled by the protocol admin. A part of its configuration defines the crowdfunding platforms listed on the Protocol: the on-chain admin account designated for operating treasuries on behalf of the platform, the platform fee, and more. Platforms are listed on the CC Protocol by the protocol admin using the `GlobalParams.enlistPlatform` method.

Once a platform is listed, it can be chosen by campaign creators in `CampaignInfoFactory.createCampaign`, and the platform admin is then able to create treasuries for the campaigns using `TreasuryFactory.deploy`.

However, platforms can also be delisted by the protocol admin through the `GlobalParams.delistPlatform` method, which does not check whether there exist unfinished campaigns/treasuries using this platform:

```
TypeScript
function delistPlatform(
    bytes32 platformHash
) external onlyOwner platformIsListed(platformHash) {
    s_platformIsListed[platformHash] = false;
    s_platformAdminAddress[platformHash] = address(0);
    s_platformFeePercent[platformHash] = 0;
    s_numberOfListedPlatforms.decrement();
    emit PlatformDelisted(platformHash);
}
```

If a platform is delisted, be it intentionally or accidentally, the existing campaigns and their treasuries can start malfunctioning, possibly locking funds pledged to them. For example, the `AllOrNothing` treasury's

`disburseFees` method, which calls into `BaseTreasury.disburseFees`, would revert on any call attempt, since the calls to `INFO.getPlatformAdminAddress` would fail:

```
TypeScript
function disburseFees()
    public
    virtual
    override
    whenCampaignNotPaused
    whenCampaignNotCancelled
{
    if (!_checkSuccessCondition()) {
        revert TreasurySuccessConditionNotFulfilled();
    }
    ...
    TOKEN.safeTransfer(
        INFO.getPlatformAdminAddress(PLATFORM_HASH),
        platformShare
    );
    ...
}
```

`CampaignInfo.getPlatformAdminAddress` just forwards the call to `GlobalParams.getPlatformAdminAddress`, which would revert with `GlobalParamsPlatformAdminNotSet`:

```
TypeScript
function getPlatformAdminAddress(
    bytes32 platformHash
)
    external
    view
    override
    platformIsListed(platformHash)
    returns (address account)
{
    account = s_platformAdminAddress[platformHash];
    if (account == address(0)) {
        revert GlobalParamsPlatformAdminNotSet(platformHash);
    }
}
```

Recommendation

Implement a graceful delisting mechanism, which doesn't immediately cause any get calls to retrieve platform information to start failing, but instead disallows creation of new campaigns and treasuries using the specified platform (in `CampaignInfoFactory.createCampaign` and `TreasuryFactory.deploy`).

This goes well with the best practice of not making any breaking changes immediately and instead introducing them with a lockup period after which the change goes live.

IMM-LOW-02

No control over expected fees during `CampaignInfoFactory.createCampaign` #3

Id	IMM-LOW-02
Severity	LOW
Category	Bug
Status	Fixed in f0ca7c769670d97ba8c86c23b4e935006e0bce00

Description

`CampaignInfoFactory.createCampaign` is meant to be used by ordinary users to create new crowdfunding campaigns using the CC Protocol and various off-chain platforms.

An economic incentive for handling campaigns is made for the Protocol and the platforms listed on it in the forms of fees paid out by the campaign treasuries once the treasury success condition is fulfilled. For transparency, the percentage of these fees is locked in at creation time.

The protocol fee is saved as an immutable value using OpenZeppelin "Clones" in `CampaignInfoFactory.createCampaign`:

```
TypeScript
function createCampaign(
    address creator,
    bytes32 identifierHash,
    bytes32[] calldata selectedPlatformHash,
    bytes32[] calldata platformDataKey,
    bytes32[] calldata platformDataValue,
    CampaignData calldata campaignData
) external override {
    ...
    bytes memory args = abi.encode(
        s_treasuryFactoryAddress,
        GLOBAL_PARAMS.getTokenAddress(),
        GLOBAL_PARAMS.getProtocolFeePercent(), // <-- Here!
        identifierHash
    );
    address clone = Clones.cloneWithImmutableArgs(s_implementation, args);
```

And the platform fees are saved in `CampaignInfo.initialize`, which is called by `CampaignInfoFactory.createCampaign` right after cloning:

```
TypeScript
function initialize(
    address creator,
    IGlobalParams globalParams,
    bytes32[] calldata selectedPlatformHash,
    bytes32[] calldata platformDataKey,
    bytes32[] calldata platformDataValue,
    CampaignData calldata campaignData
) external initializer {
    ...
    uint256 len = selectedPlatformHash.length;
    for (uint256 i = 0; i < len; ++i) {
        s_platformFeePercent[selectedPlatformHash[i]] = GLOBAL_PARAMS
            .getPlatformFeePercent(selectedPlatformHash[i]);
        s_isSelectedPlatform[selectedPlatformHash[i]] = true;
    }
```

In our perspective, there is a slight issue with this implementation: the fees are retrieved during execution time without the ability to limit them or specify the expected fees, like one usually can in other protocols (think about limiting slippage).

For correct and meaningful campaign creations, the user should be able to specify the expected fees in the `createCampaign` call, or the documentation for it should directly state that, since fees are locked in at creation, calls to `createCampaign` must be made through intermediate contracts which verify that the fee hasn't changed from the user's expectation. Link to current doc: *CampaignInfoFactory.createCampaign*.

Recommendation

Add `expectedProtocolFeePercent` and `expectedPlatformFeePercent` parameters to the `CampaignInfoFactory.createCampaign` method. Alternatively, describe this specific behaviour in the documentation.

IMM-LOW-03

Campaign owner can set arbitrary fees for `KeepWhatsRaised` #11

Id	IMM-LOW-03
Severity	LOW
Category	Bug
Status	Fixed in 9f0e5ab24503574bb96e3d3cf068f872e0604131

Description

The `KeepWhatsRaised` treasury utilizes a complex fee structure, with two separate flat fee values enacted during withdrawals, multiple gross percentage-based fees enacted during pledges, and others.

The mentioned flat fees and gross percentage-based fees are configured through custom platform data values set in the `CampaignInfo` contract linked to the treasury.

Platform data keys to be used for retrieving these fee values are controlled by the platform admin through the `KeepWhatsRaised.configureTreasury` method:

```
TypeScript
function configureTreasury(
    Config memory config,
    CampaignData memory campaignData,
    FeeKeys memory feeKeys
)
    external
    onlyPlatformAdmin(PLATFORM_HASH)
    whenCampaignNotPaused
    whenNotPaused
    whenCampaignNotCancelled
    whenNotCancelled
{
    s_config = config;
    s_feeKeys = feeKeys; // <-- fee platform data keys set here
    s_campaignData = campaignData;

    emit TreasuryConfigured(
```

```
        config,
        campaignData,
        feeKeys
    );
}
```

Keys set here are used throughout the contract in the following manner (example from `withdraw`):

```TypeScript
function withdraw(
    uint256 amount
)
    public
    currentTimeIsLess(getDeadline() + s_config.withdrawalDelay)
    whenNotPaused
    whenNotCancelled
    withdrawalEnabled
{
    uint256 flatFee = uint256(INFO.getPlatformData(s_feeKeys.flatFeeKey));
    uint256 cumulativeFee = uint256(INFO.getPlatformData(s_feeKeys.cumulativeFlatFeeKey));
    ...
}
```

`CampaignInfo.getPlatformData` simply fetches these values from the contract storage, where they're put during the campaign initialization (see `CampaignInfo.initialize`).

As it stands, the values of any platform data stored in the `CampaignInfo` contract is fully controlled by the user who calls the initial campaign creation factory method, `CampaignInfoFactory.createCampaign`. This user can well be the campaign owner themselves, since there is no access control imposed on campaign creation. The campaign owner, or the campaign creator, can set arbitrary values for valid platform data keys, and, as a consequence, arbitrary fee values for the `KeepWhatsRaised` treasury.

Since the treasury itself is created through the `TreasuryFactory.deploy` method by the platform admin at a later stage, they are able to verify that the appropriate fee values have been set in the `CampaignInfo` contract.

For this reason, we have set the vulnerability's severity to low, however we still consider it to be a security issue which can lead to the erosion of trust platforms have in the protocol, since they will have to perform thorough manual validation instead of controlling the fees themselves.

Recommendation

We recommend configuring fees directly through the `KeepWhatsRaised` treasury, which is logically owned and managed by the admin of the platform to which the treasury is linked. This makes sense, as different platforms might register the treasury, and want to set different fee values. So called "payment gateway" fees are already configured this way. The `configureTreasury` method configures the platform data keys to be used for fees, so it can be repurposed to configure the fees directly instead.

IMM-LOW-04

Refunds can drop the `AllOrNothing` treasury below the goal right before the deadline #13

Id	IMM-LOW-04
Severity	LOW
Category	Bug
Status	Acknowledged

Description

The `AllOrNothing` registry implements a pledge collection treasury which functions in a manner very close to classic Kickstarter campaigns: either the campaign reaches its funding goal and the funds are unlocked for the owner to withdraw, or the campaign reaches the deadline (or is canceled) without reaching its funding goal, in which case backers can refund their pledges.

Additionally, refunds are also available during the whole duration of the campaign, allowing it to operate smoothly without "freezing" the backer's pledges until the very end. Kickstarter provides similar functionality, but disallows pledge refunds in the last 24 hours of the campaign if they would drop the campaign below the goal (see the following article from Kickstarter: Can-I-cancel-a-pledge).

This provides an additional level of financial stability to campaigns, especially when the amount of funds collected is floating near the campaign goal. CC Protocol's `AllOrNothing` treasury does not implement any similar time locking mechanisms, allowing refunds right up until the deadline:

```
TypeScript
function claimRefund(
    uint256 tokenId
)
    external
    currentTimeIsGreater(INFO.getLaunchTime())
    whenCampaignNotPaused
    whenNotPaused
{
    if (block.timestamp >= INFO.getDeadline() && _checkSuccessCondition()) {
        revert AllOrNothingNotClaimable(tokenId);
    }
```

```
    ...
}
```

This is not a security issue which would result in the protocol functioning in an exploitable way as it is, but implementing this functionality can make the fund collection process work in a more expected and stable manner.

Recommendation

Implement additional checks in `AllOrNothing.claimRefund` which would disallow refunds when a locking period before the deadline is reached, if the requested refunds would drop the campaign below the goal.

Since this locking period might differ in duration between platforms, it can be added as a configuration option to the `AllOrNothing` treasury itself, controllable by the platform admin below campaign launch.

IMM-LOW-05

`TreasuryFactory.deploy` should check that infoAddress is registered in CampaignInfoFactory.isValidCampaignInfo #17

Id	IMM-LOW-05
Severity	LOW
Category	Bug
Status	Acknowledged Fix ready but not released in 3e89545b649f45dd659eaeb14c1343ea6ab547a7

Description

`TreasuryFactory` is one of the core contracts of the CC Protocol, meant to be used as the single trusted way of creating treasuries for campaigns. In fact, `CampaignInfo._setPlatformInfo` validates that the caller is the singleton `TreasuryFactory` instance of the protocol.

`TreasuryFactory.deploy`, the actual factory method of the contract, takes a `infoAddress` parameter, which is supposed to be the address of the `CampaignInfo` contract for which the treasury is being deployed.

However, deploy does not verify this address to be a valid CampaignInfo contract created by the `CampaignInfoFactory` counterpart. As a result, a platform admin might be tricked, or might accidentally deploy a treasury for a malicious or otherwise non-standard `CampaignInfo` contract, which would put the funds of backers using the treasury at risk.

We are not sure how the platform admins' interaction with `TreasuryFactory` will be implemented, but consider this issue to be worthwhile to fix in order to mitigate the possibility of problems arising from it.

Recommendation

`CampaignInfoFactory` contract stores a public mapping `isValidCampaignInfo`, in which the keys equal to addresses of `CampaignInfo` contracts created through `CampaignInfoFactory.createCampaign` are set to true. We propose validating the `infoAddress` parameter using this mapping.

IMM-LOW-06

Inconsistent integration of multi-platform campaigns #21

Id	IMM-LOW-06
Severity	LOW
Category	Bug
Status	Acknowledged

Description

CC Protocol considers multi-platform campaigns as one of the unique interoperability features. It is currently not one of the main features, however it is planned to be used in the long term. There are currently two separate treasury implementations, which can be used to host a campaign on different platforms with different treasury kinds: the `AllOrNothing` treasury and the `KeepWhatsRaised` treasury.

While `AllOrNothing` relies on the data configured by the owner through the main `CampaignInfo` contract, `KeepWhatsRaised` stores the campaign launch time, goal, and deadline separately. This means that, potentially, the deadlines for pledges and other processes might differ for these two treasuries of the same campaign. If the deadline set in `CampaignInfo` is earlier than the deadline set in `KeepWhatsRaised`, then pledges to the `AllOrNothing` treasury would be disallowed, but would be allowed to continue to `KeepWhatsRaised`.

The presented case is obviously not an issue, however, besides pledges, the refund period of `KeepWhatsRaised` can also intersect with the deadline of `CampaignInfo` used by `AllOrNothing`. In such case, an `AllOrNothing` treasury which passes the success condition `INFO.getTotalRaisedAmount() >= INFO.getGoalAmount()` (`_checkSuccessCondition`) at one point in time, might not pass this condition later, if a backer decides to withdraw funds from the `KeepWhatsRaised` treasury, which would be accounted in the `CampaignInfo.getTotalRaisedAmount` call.

Thus, when campaigns are hosted on multiple platforms, if configured improperly, their different stages and mechanics can conflict with one another, and lead to cases where one of the treasuries (e.g. `AllOrNothing`) is first considered to be successful, at which point `disburseFees` occurs, and then changes to and unsuccessful state, which leads to backers being able to call `claimRefund`, even though their funds have already been taxed and perhaps even withdrawn. This specific case is not a severely critical issue, since the `AllOrNothing`'s treasury's mechanisms intentionally allow pledges/withdrawals during the whole duration of the campaign, so the campaign is able to change its success at the last minute anyway. We still consider

the issue overall to be worth noting, however, and multi-platform treasury integrations should be architecturally redesigned to avoid such issues in future development.

Recommendation

Consider controlling the shared parameters of all treasuries (campaign launch time, goal, deadline) through the shared `CampaignInfo` contract, instead of storing them separately for different platforms' treasuries. Consider reworking the treasury integration to avoid treasuries' success conditions relying on one another.

IMM-INSIGHT-01

Incorrect description of the `claimFund` function #6

Id	IMM-INSIGHT-01
Severity	INSIGHT
Category	Informational
Status	Fixed in 73160f32df243626fb54c168afcd2176a46fe205

Description

The `claimFund` function in the `KeepWhatsRaised` contract is described incorrectly. The description states:

"Allows a campaign owner or authorized user to claim remaining campaign funds."

While in reality, it can only be accessed by the platform admin.

TypeScript

```
    /**
     * @dev Allows a campaign owner or authorized user to claim remaining campaign funds.
     *
     * Requirements:
     * - Claim period must have started and funds must be available.
     * - Cannot be previously claimed.
     */
    function claimFund()
        external
        onlyPlatformAdmin(PLATFORM_HASH)
        whenCampaignNotPaused
        whenNotPaused
    {
```

Recommendation

Fix documentation of function

IMM-INSIGHT-02

`CampaignInfoFactory.createCampaign` `platformDataKey` validation optimization #9

Id	IMM-INSIGHT-02
Severity	INSIGHT
Category	Gas Optimization
Status	Fixed in 7f3115f4beebb9afdf08a5e7bbf390888ec005e9

Description

`CampaignInfoFactory.createCampaign` validates nearly all arguments prior to doing any gas-costly actions such as `CampaignInfo` contract cloning and its initialization.

However, `platformDataKey` values are validated only during the `CampaignInfo.initialize` call, which occurs at a stage near the end of the `createCampaign` flow. Invalid `platformDataKey` values detected would revert the transaction, which at their current point of validation would've spent quite a lot of gas.

Of course, in the context of the Celo blockchain, on which the CC Protocol will be operating, this issue isn't as noticeable as it would be on Ethereum, for example, thanks to the lower fees. However, we still consider such optimizations important for the overall sustainability and quality of the contracts.

Recommendation

Move `platformDataKey` parameter validation to an earlier stage of the `createCampaign` flow. For example, to the beginning of the `createCampaign` method itself, where all other validation takes place.

IMM-INSIGHT-03

`CampaignInfo` lacks way to provide new platform data during `updateSelectedPlatform` #10

Id	IMM-INSIGHT-03
Severity	INSIGHT
Category	Informational
Status	Fixed in cf02e548dc5019be2335eb55719902c3d9e4fc0b

Description

`CampaignInfo` contracts are used by the CC Protocol as the main source of information about ongoing crowdfunding campaigns: they hold the campaign launch time, deadline, goal, and other parameters. In particular, `CampaignInfo` stores platform-specific data values, which can be used by other contracts in the system when working with campaigns in the context of a platform. The `KeepWhatsRaised` registry, for example, uses platform data values stored in the campaign as a source of fee information during withdrawals (see `KeepWhatsRaised.withdraw`).

Currently, the `CampaignInfo` contract allows setting up these data values during initialization through the `CampaignInfoFactory.createCampaign` method, which passes them on to `CampaignInfo.initialize`:

```
TypeScript
function initialize(
    address creator,
    IGlobalParams globalParams,
    bytes32[] calldata selectedPlatformHash,
    bytes32[] calldata platformDataKey,
    bytes32[] calldata platformDataValue,
    CampaignData calldata campaignData
) external initializer {
    ...
    for (uint256 i = 0; i < len; ++i) {
        isValid = GLOBAL_PARAMS.checkIfPlatformDataKeyValid(
            platformDataKey[i]
        );
        if (!isValid) {
```

```
            revert CampaignInfoInvalidInput();
        }
        s_platformData[platformDataKey[i]] = platformDataValue[i];
    }
}
```

When platforms are chosen during campaign creation, there is no problem with providing the platform data they require for correct operation of treasuries and other components. However, platforms can also be selected later using the `CampaignInfo.updateSelectedPlatform` method, which fails to provide a way to set new platform data values. As such, if a new platform is selected, and it turns out to require platform values to be set for treasuries to work, the user's campaign just will not function correctly with that platform. If the platform admin does their due diligence to validate the campaign configuration, they might notice the lack of necessary values and reject treasury setup for the campaign, but they might miss this issue and create the treasury anyway, in which case it might generally fail to function properly due to inconsistencies with the platform data.

Furthermore, quoting the CC Protocol team's reply to our questions on this topic, "Platform data can be completely skipped during campaign creation". As such, there must be a way for a user to set or update it later for their campaign to be correctly listed on crowdfunding platforms.

Notice that the audit report by PeckShield from May 20th, 2025 also mentioned a similar issue with `CampaignInfo.updateSelectedPlatform`: it failed to set the newly chosen platform fee, which would also lead to an inconsistent state of the `CampaignInfo` contract.

Recommendation

Allow configuring platform data during `updateSelectedPlatform` method. This will allow a user to set up the necessary data values when they choose a new platform, and to reconfigure existing values, if needed.

IMM-INSIGHT-04

`platformDataValue` values lack validation in `CampaignInfoFactory.createCampaign` #12

Id	IMM-INSIGHT-04
Severity	INSIGHT
Category	Informational
Status	Fixed in ed555d20638ec5ce6f5ae087928231987ff71bdc

Description

[`CampaignInfoFactory.createCampaign`] enforces pretty decent validation on all parameters passed to it, and it should, since the created campaign will be initialized using them. Platform-specific data values should be validated particularly thoroughly, as there is currently no way to modify them after campaign creation. Platform data keys are already properly validated during `CampaignInfo.initialize`, but the values themselves lack any sort of validation. In our opinion, one check that should be enforced is that the values are not zero, since the platform data getter, `CampaignInfo.getPlatformData`, uses zero to check if the platform data value is present:

```
TypeScript
function getPlatformData(
    bytes32 platformDataKey
) external view override returns (bytes32) {
    bytes32 platformDataValue = s_platformData[platformDataKey];
    if (platformDataValue == bytes32(0)) {
        revert CampaignInfoInvalidInput();
    }
    return platformDataValue;
}
```

Recommendation

Validate that `platformDataValue` values are not zero in `CampaignInfo.initialize` or `CampaignInfoFactory.createCampaign`, alongside other checks.

IMM-INSIGHT-05

Validate creator to be non-zero during `createCampaign` #16

Id	IMM-INSIGHT-05
Severity	INSIGHT
Category	Informational
Status	Fixed in aa2f69d0d3584f5c7fea9752251a85ed36ba69bd

Description

The CC Protocol contracts implement non-zero address checks pretty much everywhere where addresses are passed as parameters (`GlobalParams.updateProtocolAdminAddress`, `GlobalParams.enlistPlatform`, etc).

`CampaignInfo.transferOwnership` also performs this validation, albeit indirectly: the OpenZeppelin Ownable contract checks that the new owner is not zero.

```
TypeScript
function transferOwnership(address newOwner) public virtual onlyOwner {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}
```

However, the factory method `CampaignInfoFactory.createCampaign` does not validate that the initial creator address is non-zero, and passes it as is to `CampaignInfo.initialize`. `CampaignInfo.initialize`, in turn, calls the `_internal_ _transferOwnershipmethod` with `thecreator` argument, which also does not perform this check.

As a result, `CampaignInfo.transferOwnership` is inconsistent with `CampaignInfoFactory.createCampaign`, which can create a campaign and set its owner to the zero address, pretty much burning the campaign.

Recommendation

Validate the creator parameter in `CampaignInfoFactory.createCampaign` to be a non-zero address.

IMM-INSIGHT-06

Inaccurate doc for finance-related function `_calculateNetAvailable` #18

Id	IMM-INSIGHT-06
Severity	INSIGHT
Category	Informational
Status	Fixed in e8ccf17c96143d0e8c401d27c8d61175b2782419

Description

Documentation states that the `KeepWhatsRaised. _calculateNetAvailable` method calculates and accounts for the "Columbian creator tax", when applicable. In fact, it does no such thing, but it does account for the "payment gateway fee" which the documentation fails to mention.

TypeScript

```
/**
 * @dev Calculates the net available amount after deducting platform fees and applicable
taxes
 * @param pledgeId The unique identifier of the pledge.
 * @param tokenId The ID of the token representing the pledge.
 * @param pledgeAmount The total pledge amount before any deductions
 * @return The net available amount after all fees and taxes are deducted
 *
 * @notice This function performs the following calculations:
 *         1. Applies all gross percentage fees based on platform configuration
 *         2. Calculates Colombian creator tax if applicable (0.4% effective rate)
 *         3. Updates the total platform fee accumulator
 */
function _calculateNetAvailable(bytes32 pledgeId, uint256 tokenId, uint256 pledgeAmount)
internal returns (uint256) {
    uint256 totalFee = 0;

    // Gross Percentage Fee Calculation
    uint256 len = s_feeKeys.grossPercentageFeeKeys.length;
    for (uint256 i = 0; i < len; i++) {
        uint256 fee = (pledgeAmount *
uint256(INFO.getPlatformData(s_feeKeys.grossPercentageFeeKeys[i])))
                    / PERCENT_DIVIDER;
```

```
            s_platformFee += fee;
            totalFee += fee;
        }

        //Payment Gateway Fee Calculation
        uint256 paymentGatewayFee = getPaymentGatewayFee(pledgeId);
        s_platformFee += paymentGatewayFee;
        totalFee += paymentGatewayFee;

        s_tokenToPaymentFee[tokenId] = totalFee;

        return pledgeAmount - totalFee;
    }
```

We think all documentation related to finances and the accounting of various fees and taxes needs to be accurate to avoid confusing any potential users who would see this documentation.

Recommendation

Fix the doc comment to properly describe the calculations made in the `_calculateNetAvailable` method.