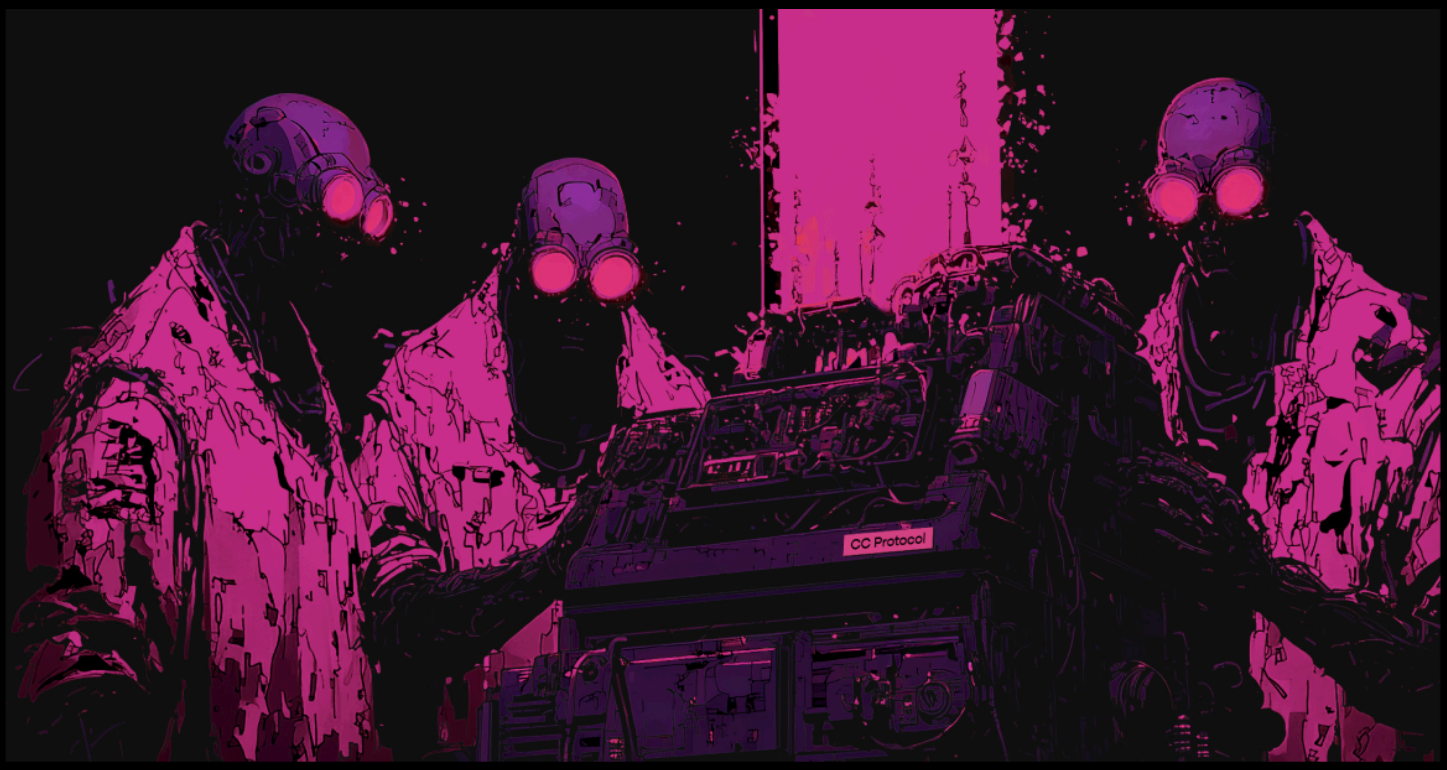

IMMUNEFI AUDIT

OakNetwork

DATE	December 10, 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, Oak Network engaged with Immunefi to review the PaymentTreasury contracts. In this period of time a total of 13 issues were identified.

SUMMARY

Name	Oak Network
Repository	https://github.com/oak-network/contracts/tree/2.0-rc, https://github.com/oak-network/safe-multisig-helper-contracts
Audit Commit	520b31fc063f6ba0614f3ce927544989c9136197, 6c8990372cd72ec3c0c443afed3ce9b1ee81ea5d
Type of Project	Infrastructure
Audit Timeline	Nov 12 - Nov 20
Fix Period	Nov 21 - Dec 4

ISSUES FOUND

Severity	Count	Fixed	Acknowledged
Critical	1	1	0
High	3	3	0
Medium	2	2	0
Low	4	4	0
INSIGHT	3	3	0

CATEGORY BREAKDOWN

Bug	11
Gas Optimization	1
Informational	1

FINDINGS

IMM-CRIT-01

Incompatible function signatures in PaymentTreasuryAdapter #10

Id	IMM-CRIT-01
Severity	Critical
Category	Bug
Status	Fixed in `01fcd39fe4cb8b863324910ba35bb7c28d00a1fc`

Description

The `PaymentTreasuryAdapter` uses outdated function signatures that are incompatible with the current `BasePaymentTreasury` implementation, causing all adapter calls to fail.

The `createPayment()` function in the adapter encodes only 5 parameters:

```
TypeScript
bytes memory data = abi.encodeWithSignature(
    "createPayment(bytes32,bytes32,bytes32,uint256,uint256)",
    paymentId,
    buyerId,
    itemId,
    amount,
    expiration
);
```

However, the actual treasury function requires 8 parameters including `paymentToken`, `lineItems[]`, and `externalFees[]`:

```typescript
function createPayment(
    bytes32 paymentId,
    bytes32 buyerId,
    bytes32 itemId,
    address paymentToken,  // Missing in adapter
    uint256 amount,
    uint256 expiration,
    ICampaignPaymentTreasury.LineItem[] calldata lineItems,  // Missing in adapter
    ICampaignPaymentTreasury.ExternalFees[] calldata externalFees  // Missing in adapter
) public override virtual onlyPlatformAdmin(PLATFORM_HASH)
```

Similarly, `confirmPayment()` is missing the `buyerAddress` parameter, and `confirmPaymentBatch()` is missing the `buyerAddresses[]` array. When these adapter functions are called, the treasury contract receives incorrectly encoded calldata, causing function selector mismatch and immediate reversion with `CallFailed()`.

Recommendation

Update function signatures in `PaymentTreasuryAdapter.sol`:

1. `createPayment()` - Add missing parameters:

```typescript
// Add after itemId parameter:
address paymentToken,

// Add after expiration parameter:
IPaymentTreasury.LineItem[] calldata lineItems,
IPaymentTreasury.ExternalFees[] calldata externalFees

// Update abi.encodeWithSignature (line 17-23):
bytes memory data = abi.encodeWithSignature(
```

```
"createPayment(bytes32,bytes32,bytes32,address,uint256,uint256,(bytes32,uint256)[],(address,uint256)[])",
    paymentId, buyerId, itemId, paymentToken, amount, expiration, lineItems, externalFees
);
```

2. `confirmPayment()` - Add buyerAddress parameter:

```
TypeScript
// Add after paymentId parameter:
address buyerAddress

// Update abi.encodeWithSignature (line 53-55):
bytes memory data = abi.encodeWithSignature(
    "confirmPayment(bytes32,address)",
    paymentId, buyerAddress
);
```

3. `confirmPaymentBatch()` - Add buyerAddresses array:

```
TypeScript
// Add after paymentIds parameter:
address[] calldata buyerAddresses

// Update abi.encodeWithSignature (line 69-71):
bytes memory data = abi.encodeWithSignature(
    "confirmPaymentBatch(bytes32[],address[])",
    paymentIds, buyerAddresses
);
```

IMM-HIGH-01

Blocking createPayment and other payments via front-running processCryptoPayment #8

Id	IMM-HIGH-01
Severity	High
Category	Bug
Status	Fixed in 45d049c75d58b31a8b568d7c5eb44d5b773bcd1f and d3f37b0b045d37dd71cafe3e5f0003707a9e145d

Description

All payment-related functions (important ones are `createPayment`, `createPaymentBatch`, `processCryptoPayment`) accept a user-controlled `paymentId` parameter that is validated to be unique. For external payments, this value comes from payment providers such as Stripe. The onchain `processCryptoPayment`, however, also uses the same `paymentId` system, which allows attackers to front-run legitimate on- and off-chain transactions and block them by using the same payment ID.

TypeScript

```
// All these functions check payment ID uniqueness:
function createPayment(bytes32 paymentId, ...) { ... }
function createPaymentBatch(bytes32 calldata []paymentIds, ...) { ... }
function processCryptoPayment(bytes32 paymentId, ...) { ... }

// Common check in all functions:
if(s_payment[paymentId].buyerId != ZERO_BYTES || s_payment[paymentId].buyerAddress !=
address(0)){
    revert PaymentTreasuryPaymentAlreadyExist(paymentId);
}
```

The `paymentId` is not scoped to `msg.sender`. An attacker can monitor the mempool for any pending payment transaction, extract the `paymentId`, and front-run it:

- Front-run `createPayment` with `processCryptoPayment(sameId, ..., 1 wei, ...)`
- Front-run `processCryptoPayment` with another `processCryptoPayment(sameId, ..., 1 wei, ...)`

When the legitimate transaction executes, it reverts with `PaymentTreasuryPaymentAlreadyExist`, effectively blocking all real payments to any campaign.

This vulnerability is particularly severe for Stripe integration. An attacker can monitor the platform admin's `createPayment(stripePaymentId, ...)` transactions in the mempool, extract Stripe payment IDs, and front-run them with on-chain `processCryptoPayment(stripePaymentId, ..., 1 wei, ...)`.

The payment ID becomes occupied by attacker's 1 wei transaction if the front-run attack is successful.
When an off-chain payment's migration to the chain via `createPayment` transaction arrives, it will fail because the ID is already taken. This can severly damage the Stripe integration - legitimate off-chain payments cannot be recorded on-chain, breaking other off-chain systems relying on this and causing potential user fund loss, or at the very least resulting in the need to process refunds for all these failed payment creations.

This is a recurrence of the same vulnerability pattern that previously existed in `KeepWhatsRaised` with `pledgeId` parameters.

Recommendation

Scope the payment ID to the caller by hashing it with `_msgSender()` in `processCryptoPayment` and other on-chain payment processing functions, and with the zero address for off-chain payment methods (which will allow the platform admin address to change in between these calls without breaking existing payment treasuries), similar to the same fix applied to `KeepWhatsRaised` in commit 1d6ad87:

```
TypeScript
function createPayment(bytes32 paymentId, ...) {
    bytes32 internalPaymentId = keccak256(abi.encodePacked(paymentId, ZERO_ADDRESS));
```

```
    if(s_payment[internalPaymentId].buyerId != ZERO_BYTES){
        revert PaymentTreasuryPaymentAlreadyExist(internalPaymentId);
    }
    s_payment[internalPaymentId] = PaymentInfo({...});

    ...
}

function processCryptoPayment(bytes32 paymentId, ...) {
    bytes32 internalPaymentId = keccak256(abi.encodePacked(paymentId, _msgSender()));
    ...
}
```

This preserves the external `paymentId` for off-chain tracking with global uniqueness while ensuring uniqueness per caller on-chain.

IMM-HIGH-02

Missing Access Control on withdraw allows blocking refund mechanism #9

Id	IMM-HIGH-02
Severity	High
Category	Bug
Status	Fixed in 6a0f9d89c5675b783bcbb848a9940227f8f17fd6

Description

The `withdraw()` function has no access control and can be called by anyone. Combined with `PaymentTreasury._checkSuccessCondition()` always returning `true`, this allows any attacker to force withdrawals immediately after payments are received, breaking the entire refund mechanism.

```TypeScript
function withdraw()
    public  // No access control - anyone can call
    virtual
    override
    whenCampaignNotPaused
    whenCampaignNotCancelled
{
    if (!_checkSuccessCondition()) {  // Always returns true in PaymentTreasury
        revert PaymentTreasurySuccessConditionNotFulfilled();
    }

    address recipient = INFO.owner();
    // ...
    for (uint256 i = 0; i < acceptedTokens.length; i++) {
        address token = acceptedTokens[i];
        uint256 balance = s_availableConfirmedPerToken[token];
```

```
        if (balance > 0) {
            // Transfer funds to owner
            s_availableConfirmedPerToken[token] = 0;  // Reset to zero
            IERC20(token).safeTransfer(recipient, withdrawalAmount);
        }
    }
}
```

When `withdraw()` is called, `s_availableConfirmedPerToken[token]` is reset to zero. However, `claimRefund()` checks this value to determine if refunds are possible:

TypeScript

```
function claimRefund(bytes32 paymentId, address refundAddress) public {
    uint256 availablePaymentAmount = s_availableConfirmedPerToken[paymentToken];

    if (amountToRefund == 0 || availablePaymentAmount < amountToRefund) {
        revert PaymentTreasuryPaymentNotClaimable(paymentId);  // Will always revert after
withdraw
    }
    // ...
}
```

Attack scenario:

1. User makes any payment (via `processCryptoPayment()` or `createPayment()` + `confirmPayment()`)
2. Attacker immediately calls `withdraw()`
3. Campaign owner receives the funds, `s_availableConfirmedPerToken[token]` becomes 0
4. User later tries to claim refund
5. Refund fails with `PaymentTreasuryPaymentNotClaimable` because available balance is 0

This completely breaks the refund mechanism for all payment types. Any attacker can force premature

withdrawals after every payment, preventing all future refunds regardless of the campaign owner's intent. The campaign owner has no way to prevent this attack or return funds collected through non-crypto payments (via createPayment & confirmPayment) except by converting crypto to fiat and manually transferring back to the buyer's bank account.

This creates legal liability issues for the platform if it provides any refund guarantees to users, as the smart contract-based refund mechanism can be completely disabled by any third party.

Recommendation

Restrict `withdraw()` to be callable only by the campaign owner, or alternatively allow both the campaign owner and platform admin (following the same pattern as `KeepWhatsRaised`).

Add the `onlyPlatformAdminOrCampaignOwner` modifier (similar to `KeepWhatsRaised`):

```
TypeScript
// Add this modifier to BasePaymentTreasury
modifier onlyPlatformAdminOrCampaignOwner() {
    if (
        _msgSender() != INFO.getPlatformAdminAddress(PLATFORM_HASH) &&
        _msgSender() != INFO.owner()
    ) {
        revert AccessCheckerUnauthorized();
    }
    _;
}

function withdraw()
    public
    virtual
    override
    onlyPlatformAdminOrCampaignOwner  // Apply this modifier
    whenCampaignNotPaused
```

```
    whenCampaignNotCancelled
{
    if (!_checkSuccessCondition()) {
        revert PaymentTreasurySuccessConditionNotFulfilled();
    }

    // ...
}
```

This prevents arbitrary third parties from forcing premature withdrawals while allowing authorized parties (owner or platform admin) to control fund withdrawals.

IMM-HIGH-03

Campaign owner can block protocol/platform fee & reward collection in PaymentTreasury and TimeConstrainedPaymentTreasury #13

Id	IMM-HIGH-03
Severity	High
Category	Bug
Status	Fixed in `18f697b95ac9bad47cb0f50138763167d7ecb797`

Description

Campaign owners, who we can consider to be low-privileged users in Oak Network, are able to cancel their campaign through `CampaignInfo._cancelCampaign` at any point in time, including after launch and before the deadline. Since `BasePaymentTreasury` currently enforces the `whenCampaignNotPaused` and `whenCampaignNotCancelled` modifiers for all operations, including fee & reward claim methods intended for the platform/protocol (`disburseFees`, `claimNonGoalLineItems`, `claimExpiredFunds`), they can effectively block the platform they're using to host their campaign, and Oak Network itself, from being compensated according to the configured fee structure.

Besides canceling the `CampaignInfo`, owners are also able to cancel the treasuries themselves in the cases of `PaymentTreasury` and `TimeConstrainedPaymentTreasury`, both of which override `cancelTreasury` like this:

```
TypeScript
function cancelTreasury(bytes32 message) public override {
    if (
        _msgSender() != INFO.getPlatformAdminAddress(PLATFORM_HASH) &&
        _msgSender() != INFO.owner()
    ) {
        revert PaymentTreasuryUnAuthorized();
```

```
    }
    _cancel(message);
}
```

Since `PaymentTreasury` enforces the treasury-bound `whenNotPaused` and `whenNotCancelled` modifiers for the `claimExpiredFunds` and `disburseFees` methods, they are affected through the treasury cancelations additionally to `CampaignInfo` cancelations.

This issue is especially serious due to the fact that `withdraw` calls for the payment treasuries are not time-locked and don't require explicit approval, contrary to withdrawals in the `KeepWhatsRaised` treasury (`KeepWhatsRaised.withdraw` requires the platform admin to call `approveWithdrawal`). A malicious campaign owner can wait for enough rewards to accumulate close to the campaign deadline, withdraw them, and cancel the treasury or campaign to lock the remaining protocol and platform fees and rewards.

Recommendation

It should not be possible to disable protocol and platform fee claiming completely, freezing the funds on the treasury contract forever. The `KeepWhatsRaised` treasury, for example, explicitly allows fund claims despite any present cancelations, blocking only when the treasury or campaign is "paused". Example from `KeepWhatsRaised.claimFund`:

```TypeScript
function claimFund()
    external
    onlyPlatformAdmin(PLATFORM_HASH)
    whenCampaignNotPaused
    whenNotPaused
{
    bool isCancelled = s_cancellationTime > 0;
    uint256 cancelLimit = s_cancellationTime + s_config.refundDelay;
    uint256 deadlineLimit = getDeadline() + s_config.withdrawalDelay;
```

```
    if ((isCancelled && block.timestamp <= cancelLimit) || (!isCancelled && block.timestamp
<= deadlineLimit)) {
        revert KeepWhatsRaisedNotClaimableAdmin();
    }

    ...

}
```

IMM-MED-01

Missing treasury state validation in TimeConstrainedPaymentTreasury #7

Id	IMM-MED-01
Severity	Medium
Category	Bug
Status	Fixed in b35f343d0e92e51447b5d7cbd7a074ad9afc785f

Description

The `TimeConstrainedPaymentTreasury` contract uses wrong modifiers on its overridden functions, causing the treasury's own pause and cancel state to never be checked. All functions use `whenCampaignNotPaused` and `whenCampaignNotCancelled` which only validate the campaign's state, not the treasury's state.

For example, in `TimeConstrainedPaymentTreasury`:

```
TypeScript
function createPayment(...) public override whenCampaignNotPaused whenCampaignNotCancelled {
    _checkTimeWithinRange();
    super.createPayment(...);  // Already has whenCampaignNotPaused whenCampaignNotCancelled
}
```

The base implementation in `BasePaymentTreasury` already includes these campaign state modifiers:

```
TypeScript
function createPayment(...) public override virtual onlyPlatformAdmin(PLATFORM_HASH)
whenCampaignNotPaused whenCampaignNotCancelled {
    // implementation
}
```

This means the treasury's own pause/cancel state is never validated. If the treasury itself is paused or

cancelled through `PausableCancellable` functions, all payment operations will still execute because only the campaign state is checked, not the treasury state. The same issue affects all overridden functions: `createPaymentBatch()`, `processCryptoPayment()`, `cancelPayment()`, `confirmPayment()`, `confirmPaymentBatch()`, `claimRefund()`, `claimExpiredFunds()`, `disburseFees()`, and `withdraw()`.

In contrast, `PaymentTreasury` correctly uses `whenNotPaused` and `whenNotCancelled` (without "Campaign") which check the treasury's own state from `PausableCancellable`, allowing proper treasury-level pause/cancel functionality.

Recommendation

Replace `whenCampaignNotPaused` and `whenCampaignNotCancelled` with `whenNotPaused` and `whenNotCancelled` in all overridden functions in `TimeConstrainedPaymentTreasury.sol` to match the pattern used in `PaymentTreasury.sol`:

```
TypeScript
function createPayment(...) public override whenNotPaused whenNotCancelled {
    _checkTimeWithinRange();
    super.createPayment(...);
}
```

This ensures the treasury's own pause/cancel state is properly checked before calling the base implementation, which will then check the campaign's state. Without this fix, pausing or cancelling the treasury contract directly has no effect on payment operations.

IMM-MED-02

Treasuries missing EIP-2771 Meta-Transaction Support #12

Id	IMM-MED-02
Severity	Medium
Category	Bug
Status	Fixed in 0807ac0f5019232337760e7379ffcf1e924c028e

Description

None of the treasury contracts (`BaseTreasury`, `BasePaymentTreasury`, `AllOrNothing`, `KeepWhatsRaised`, `PaymentTreasury`, `TimeConstrainedPaymentTreasury`) implement meta-transaction sender extraction, making the entire safe-multisig-helper-contracts adapter integration non-functional.

All adapter functions in `PaymentTreasuryAdapter.sol` correctly append `msg.sender` to calldata following the EIP-2771 pattern:

```
TypeScript
bytes memory dataWithSender = abi.encodePacked(data, msg.sender);
(bool success, ) = treasury.call(dataWithSender);
```

However, `BasePaymentTreasury` never extracts this appended sender. It inherits from `CampaignAccessChecker` which uses OpenZeppelin's `Context._msgSender()` that simply returns `msg.sender` (the adapter contract address), not the actual Safe multisig address appended in calldata.

This causes all access control checks to see the `AdapterManager` contract address instead of the Safe multisig address. When the Safe calls `AdapterManager.createPayment()`, which then calls `treasury.createPayment()`, the treasury's `onlyPlatformAdmin` modifier checks `_msgSender()` and gets `address(AdapterManager)` instead of the Safe's address. Since `address(AdapterManager) != platformAdmin`, every transaction reverts with `AccessCheckerUnauthorized()`.

The issue is confirmed by `MockTreasury.sol` in the adapter repository, which correctly implements sender extraction for testing. This implementation was never ported to the production treasury contracts, making the entire adapter system unusable for any function with access control modifiers (`onlyPlatformAdmin`, `onlyCampaignOwner`, `onlyProtocolAdmin`).

Recommendation

Override `_msgSender()` in both `BaseTreasury` and `BasePaymentTreasury` to extract the sender from calldata only when called by a trusted forwarder (adapter contract). Simply extracting from calldata unconditionally would allow any attacker to append a fake address and bypass access control.

Implement EIP-2771 trusted forwarder pattern in both `BaseTreasury` and `BasePaymentTreasury`:

```typescript
// Add to BasePaymentTreasury storage
address public trustedForwarder;

// Set during initialization
function initialize(..., address _trustedForwarder) external initializer {
    // ... existing initialization
    trustedForwarder = _trustedForwarder;
}

// Override _msgSender to extract sender only from trusted forwarder
function _msgSender() internal view override returns (address sender) {
    if (msg.sender == trustedForwarder && msg.data.length >= 20) {
        assembly {
            sender := shr(96, calldataload(sub(calldatasize(), 20)))
        }
    } else {
        sender = msg.sender;
    }
}
```

This ensures that only calls from the designated adapter contract can specify the original sender, while

direct calls to the treasury still use `msg.sender` for access control. Without the trusted forwarder check, any attacker could call the treasury directly with an appended admin address and bypass all access restrictions.

IMM-LOW-01

JSON Injection in NFT Metadata #1

Id	IMM-LOW-01
Severity	LOW
Category	Bug
Status	Fixed in 37408ece3b389acd1834a105079c50aa096ad6ad

Description

The `tokenURI()` function generates NFT metadata on-chain by concatenating user-controlled strings directly into JSON without escaping special characters. Campaign owners control the NFT name through `initialize()` and can set `imageURI` via `setImageURI()`.

If the NFT name or image URI contains quotes (`"`), backslashes (`\`), or control characters (`\n`, `\r`, `\t`), the resulting JSON becomes invalid or malformed. This breaks NFT metadata on marketplaces.

TypeScript

```
function tokenURI(uint256 tokenId) public view virtual override returns (string memory) {
    _requireOwned(tokenId);
    PledgeData memory data = s_pledgeData[tokenId];

    string memory json = string(
        abi.encodePacked(
            '{"name":"', name(), " #", tokenId.toString(),  // No escaping
            '","image":"', s_imageURI,                      // No escaping
            '","attributes":[',
            // ...
            "]}"
        )
    );
```

```
    return string(abi.encodePacked("data:application/json;base64,",
Base64.encode(bytes(json))));
}
```

Example attack: Setting NFT name to `Pledge NFT","malicious":"injected` produces:

```
TypeScript
{"name":"Pledge NFT","malicious":"injected #1","image":"..."}
```

The JSON structure is altered, and if more complex injection is used (like unescaped newlines), the JSON becomes completely invalid causing `JSON.parse()` failures in all frontend applications and marketplaces.

Additionally, this on-chain metadata generation is extremely gas-intensive. The function performs multiple `abi.encodePacked()` calls, `toString()` conversions, `toHexString()` operations, and Base64 encoding, consuming approximately 80,000-150,000 gas per call. This is 40-75x more expensive than standard off-chain metadata approaches which cost around 1,500-2,000 gas.

Recommendation

Switch to off-chain metadata storage with a base URI pattern. Add a `baseTokenURI` storage variable and modify `tokenURI()` to return `baseTokenURI + tokenId + ".json"`. This reduces gas costs by ~98% and allows proper JSON generation on the backend where escaping can be handled correctly.

If on-chain metadata must be preserved, implement JSON string escaping that handles quotes, backslashes, and control characters before concatenation. However, this will add another 20,000-50,000 gas per call.

At minimum, add validation in `initialize()` to reject NFT names and image URIs containing quotes, backslashes, or control characters below `0x20`.

IMM-LOW-02

Missing launch buffer validation in launch time update #5

Id	IMM-LOW-02
Severity	LOW
Category	Bug
Status	Fixed in d9fd1a172745bad12015f1d01ced47ab4578f581

Description

The `updateLaunchTime()` function allows the campaign owner to set a launch time without enforcing the `CAMPAIGN_LAUNCH_BUFFER` that is required during initial campaign creation. This allows the owner to bypass the buffer period intended to give backers and platforms adequate preparation time.

During campaign creation in `CampaignInfoFactory.createCampaign()`, the launch buffer is enforced:

```
TypeScript
uint256 campaignLaunchBuffer =
uint256(globalParams.getFromRegistry(DataRegistryKeys.CAMPAIGN_LAUNCH_BUFFER));
if (campaignData.launchTime < block.timestamp + campaignLaunchBuffer) {
    revert CampaignInfoFactoryInvalidInput();
}
```

However, `updateLaunchTime()` only checks that the new time is in the future:

```
TypeScript
function updateLaunchTime(uint256 launchTime) external {
    if (launchTime < block.timestamp || getDeadline() <= launchTime) {
        revert CampaignInfoInvalidInput();
    }
    s_campaignData.launchTime = launchTime;
}
```

This allows a campaign owner to update the launch time to be only 1 second in the future, circumventing the buffer requirement. This can be exploited to launch campaigns with insufficient notice, preventing backers from making informed decisions or platforms from properly preparing campaign listings.

Recommendation

Add the same buffer validation used during campaign creation:

```TypeScript
function updateLaunchTime(uint256 launchTime) external {
    uint256 campaignLaunchBuffer =
uint256(_getGlobalParams().getFromRegistry(DataRegistryKeys.CAMPAIGN_LAUNCH_BUFFER));

    if (launchTime < block.timestamp + campaignLaunchBuffer || getDeadline() <= launchTime) {
        revert CampaignInfoInvalidInput();
    }
    s_campaignData.launchTime = launchTime;
}
```

This ensures consistent enforcement of the launch buffer regardless of whether the time is set during creation or updated later.

IMM-LOW-03

Missing minimum duration validation in deadline update #6

Id	IMM-LOW-03
Severity	LOW
Category	Bug
Status	Fixed in 79aab5810fa9a33f74201eb41143a50c9711653a

Description

The `updateDeadline()` function allows the campaign owner to set a deadline without enforcing the `MINIMUM_CAMPAIGN_DURATION` that is required during initial campaign creation. This allows the owner to bypass the minimum duration requirement intended to ensure campaigns run for a reasonable timeframe.

During campaign creation in `CampaignInfoFactory.createCampaign()`, the minimum duration is enforced:

```
TypeScript
uint256 minimumCampaignDuration =
uint256(globalParams.getFromRegistry(DataRegistryKeys.MINIMUM_CAMPAIGN_DURATION));
if (campaignData.deadline < campaignData.launchTime + minimumCampaignDuration) {
    revert CampaignInfoFactoryInvalidInput();
}
```

However, `updateDeadline()` only checks that the deadline is after the launch time:

```
TypeScript
function updateDeadline(uint256 deadline) external {
    if (deadline <= getLaunchTime()) {
        revert CampaignInfoInvalidInput();
    }
    s_campaignData.deadline = deadline;
}
```

This allows a campaign owner to update the deadline to be only 1 second after the launch time, circumventing the minimum duration requirement. This can be exploited to create artificially short campaigns that pressure backers into making rushed decisions without adequate time to evaluate the campaign.

Recommendation

Add the same minimum duration validation used during campaign creation:

```TypeScript
function updateDeadline(uint256 deadline) external {
    uint256 minimumCampaignDuration =
uint256(_getGlobalParams().getFromRegistry(DataRegistryKeys.MINIMUM_CAMPAIGN_DURATION));

    if (deadline <= getLaunchTime() || deadline < getLaunchTime() + minimumCampaignDuration)
{
        revert CampaignInfoInvalidInput();
    }
    s_campaignData.deadline = deadline;
}
```

This ensures consistent enforcement of the minimum campaign duration regardless of whether the deadline is set during creation or updated later.

IMM-LOW-04

Using encodeWithSignature Instead of encodeCall #11

Id	IMM-LOW-04
Severity	LOW
Category	Bug
Status	Fixed in 01fcd39fe4cb8b863324910ba35bb7c28d00a1fc and c38f75a11d2ba6f57fff6eecd37dee03d7ea8f11

Description

All 24 adapter functions use `abi.encodeWithSignature()` with string literals instead of `abi.encodeCall()`. This provides zero type safety - typos in function names or parameter types compile successfully but fail at runtime.

Issues:

- Typos in function names not caught by compiler
- Wrong parameter types compile fine, call wrong function or fail silently
- No validation that function exists on target contract
- Changes to treasury interfaces don't trigger compile errors in adapters

Affected: All 24 functions in `PaymentTreasuryAdapter.sol`, `KeepWhatsRaisedAdapter.sol`, `AllOrNothingAdapter.sol`.

Current unsafe pattern (`cancelPayment` example):

```
TypeScript
bytes memory data = abi.encodeWithSignature(
    "cancelPayment(bytes32)",  // String literal - no validation
    paymentId
);
```

Usage of `abi.encodeWithSignature` leads to silent failures, wrong function calls, broken integrations after upgrades.

Recommendation

Replace all `abi.encodeWithSignature()` calls with `abi.encodeCall()` to get compile-time type safety. The compiler will validate function names, parameter types, and function existence, preventing typos and signature mismatches that only fail at runtime.

Current unsafe code:

```
TypeScript
bytes memory data = abi.encodeWithSignature(
    "cancelPayment(bytes32)",
    paymentId
);
```

Should be replaced with:

```
TypeScript
bytes memory data = abi.encodeCall(
    IPaymentTreasury.cancelPayment,
    (paymentId)
);
```

This change needs to be applied to all 24 adapter functions: 9 functions in `PaymentTreasuryAdapter.sol` using `IPaymentTreasury`, 12 functions in `KeepWhatsRaisedAdapter.sol` using `IKeepWhatsRaised`, and 3 functions in `AllOrNothingAdapter.sol` using `IAllOrNothing`.

With `abi.encodeCall()`, the compiler catches typos, validates parameter types, confirms function existence, and automatically detects when treasury interfaces change. This eliminates an entire class of bugs that currently compile successfully but fail silently at runtime.

IMM-INSIGHT-01

Event order confusion after in NFT Minting #2

Id	IMM-INSIGHT-01
Severity	INSIGHT
Category	Informational
Status	Fixed in cedf1f643aa68e2e5be88d0944e4a572299ad052

Description

The `mintNFTForPledge()` function emits the `PledgeNFTMinted` event after calling `_safeMint()`. Since `_safeMint()` triggers the `onERC721Received()` callback on the recipient, a reentrancy can occur where the NFT already exists and external calls can be made before the event is emitted.

```
TypeScript
function mintNFTForPledge(
    address backer,
    bytes32 reward,
    address tokenAddress,
    uint256 amount,
    uint256 shippingFee,
    uint256 tipAmount
) public virtual onlyRole(MINTER_ROLE) returns (uint256 tokenId) {
    s_tokenIdCounter.increment();
    tokenId = s_tokenIdCounter.current();

    s_pledgeData[tokenId] = PledgeData({...});

    _safeMint(backer, tokenId);  // Calls onERC721Received - reentrancy point

    emit PledgeNFTMinted(tokenId, backer, msg.sender, reward);  // Event after callback
```

```
        return tokenId;
    }
```

This breaks the expected order for off-chain indexers that rely on events to track state changes. During the `onERC721Received()` callback, the receiver can call back into the contract (like `withdraw()` or `disburseFees()` as shown in existing reentrancy tests) and emit other events before `PledgeNFTMinted` is recorded. Indexers will see these events out of order, potentially causing incorrect state tracking.

Recommendation

Emit the `PledgeNFTMinted` event before calling `_safeMint()` to ensure correct chronological event ordering. The OpenZeppelin ERC721 `Transfer` event will still be emitted during `_safeMint()` after the custom event, which is the expected pattern.

Alternatively, add a `nonReentrant` modifier to `mintNFTForPledge()` to prevent any callbacks from executing during minting, though this adds gas costs and doesn't solve the fundamental event ordering issue.

IMM-INSIGHT-02

Contract reference fetched inside loop #3

Id	IMM-INSIGHT-02
Severity	INSIGHT
Category	Gas Optimization
Status	Fixed in 68b7ca08a22785bbc8e7c54aa002f87a01d72d55

Description

The `updateSelectedPlatform()` function calls `_getGlobalParams()` inside a loop. Each call loads the global params contract address from storage and performs an external call.

```
TypeScript
for (uint256 i = 0; i < platformDataKey.length; i++) {
    isValid = _getGlobalParams().checkIfPlatformDataKeyValid(  // External call in loop
        platformDataKey[i]
    );
}
```

Each `_getGlobalParams()` call reads from storage and returns a contract reference. While the external call cost dominates, the repeated storage read adds unnecessary overhead. For loops with many platform data keys, this accumulates significantly.

Recommendation

Cache `_getGlobalParams()` before the loop:

```
TypeScript
IGlobalParams globalParams = _getGlobalParams();
for (uint256 i = 0; i < platformDataKey.length; i++) {
    isValid = globalParams.checkIfPlatformDataKeyValid(platformDataKey[i]);
```

```
}
```

This eliminates repeated storage reads and saving gas.

IMM-INSIGHT-03

Precision loss in reward price denormalization #4

Id	IMM-INSIGHT-03
Severity	INSIGHT
Category	Bug
Status	Fixed in 13bfbe10fc803a6c6cce5de329ddf4490e3ae3ec

Description

The `_denormalizeAmount()` function rounds down when converting from 18 decimals to token decimals, causing backers to pay slightly less than the intended reward price. This occurs in `AllOrNothing._pledge()` and `KeepWhatsRaised._pledge()` when processing reward pledges.

```
TypeScript
function _denormalizeAmount(address token, uint256 amount) internal view returns (uint256) {
    uint8 decimals = IERC20Metadata(token).decimals();

    if (decimals < STANDARD_DECIMALS) {
        return amount / (10 ** (STANDARD_DECIMALS - decimals));  // Rounds down
    }
    // ...
}
```

When reward prices are stored in 18 decimals and converted to tokens with fewer decimals like USDC (6 decimals), the division truncates fractional amounts. For example:

- Reward price: `10.123456789 USDC` = `10123456789000000000` (18 decimals)
- After denormalization: `10123456789000000000 / 10^12` = `10123456` (10.123456 USDC)
- Lost amount: `0.000000789 USDC`

The campaign owner loses the fractional difference on every pledge. While the individual loss per

transaction is minimal, the fix is straightforward and should be implemented to ensure accurate pricing.

Recommendation

Round up instead of down to ensure the campaign owner receives at least the intended amount:

```TypeScript
function _denormalizeAmount(address token, uint256 amount) internal view returns (uint256) {
    uint8 decimals = IERC20Metadata(token).decimals();

    if (decimals < STANDARD_DECIMALS) {
        uint256 divisor = 10 ** (STANDARD_DECIMALS - decimals);
        return (amount + divisor - 1) / divisor;  // Round up
    }
    // ...
}
```

Alternatively, validate during reward creation that prices convert cleanly without remainders when using tokens with fewer decimals.